

TIBCO® Data Virtualization

Reference Guide

Version 8.1

Last Updated: March 25, 2019

Important Information

SOME TIBCO SOFTWARE EMBEDS OR BUNDLES OTHER TIBCO SOFTWARE. USE OF SUCH EMBEDDED OR BUNDLED TIBCO SOFTWARE IS SOLELY TO ENABLE THE FUNCTIONALITY (OR PROVIDE LIMITED ADD-ON FUNCTIONALITY) OF THE LICENSED TIBCO SOFTWARE. THE EMBEDDED OR BUNDLED SOFTWARE IS NOT LICENSED TO BE USED OR ACCESSED BY ANY OTHER TIBCO SOFTWARE OR FOR ANY OTHER PURPOSE.

USE OF TIBCO SOFTWARE AND THIS DOCUMENTATION IS SUBJECT TO THE TERMS AND CONDITIONS OF A LICENSE AGREEMENT FOUND IN EITHER A SEPARATELY EXECUTED SOFTWARE LICENSE AGREEMENT, OR, IF THERE IS NO SUCH SEPARATE AGREEMENT, THE CLICKWRAP END USER LICENSE AGREEMENT WHICH IS DISPLAYED DURING DOWNLOAD OR INSTALLATION OF THE SOFTWARE (AND WHICH IS DUPLICATED IN THE LICENSE FILE) OR IF THERE IS NO SUCH SOFTWARE LICENSE AGREEMENT OR CLICKWRAP END USER LICENSE AGREEMENT, THE LICENSE(S) LOCATED IN THE "LICENSE" FILE(S) OF THE SOFTWARE. USE OF THIS DOCUMENTATION IS SUBJECT TO THOSE TERMS AND CONDITIONS, AND YOUR USE HEREOF SHALL CONSTITUTE ACCEPTANCE OF AND AN AGREEMENT TO BE BOUND BY THE SAME.

This document contains confidential information that is subject to U.S. and international copyright laws and treaties. No part of this document may be reproduced in any form without the written authorization of TIBCO Software Inc.

TIBCO and the TIBCO logo are either registered trademarks or trademarks of TIBCO Software Inc. in the United States and/or other countries

TIBCO, Two-Second Advantage, TIBCO Spotfire, TIBCO ActiveSpaces, TIBCO Spotfire Developer, TIBCO EMS, TIBCO Spotfire Automation Services, TIBCO Enterprise Runtime for R, TIBCO Spotfire Server, TIBCO Spotfire Web Player, TIBCO Spotfire Statistics Services, S-PLUS, and TIBCO Spotfire S+ are either registered trademarks or trademarks of TIBCO Software Inc. in the United States and/or other countries.

All other product and company names and marks mentioned in this document are the property of their respective owners and are mentioned for identification purposes only.

THIS SOFTWARE MAY BE AVAILABLE ON MULTIPLE OPERATING SYSTEMS. HOWEVER, NOT ALL OPERATING SYSTEM PLATFORMS FOR A SPECIFIC SOFTWARE VERSION ARE RELEASED AT THE SAME TIME. SEE THE README FILE FOR THE AVAILABILITY OF THIS SOFTWARE VERSION ON A SPECIFIC OPERATING SYSTEM PLATFORM.

THIS DOCUMENTATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

THIS DOCUMENTATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THIS DOCUMENTATION. TIBCO SOFTWARE INC. MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THIS DOCUMENTATION AT ANY TIME.

THE CONTENTS OF THIS DOCUMENTATION MAY BE MODIFIED AND/OR QUALIFIED, DIRECTLY OR INDIRECTLY, BY OTHER DOCUMENTATION WHICH ACCOMPANIES THIS SOFTWARE, INCLUDING BUT NOT LIMITED TO ANY RELEASE NOTES AND "READ ME" FILES.

Copyright © 2004-2019 TIBCO Software Inc. ALL RIGHTS RESERVED.

TIBCO Software Inc. Confidential Information

Contents

| | |
|---|-----------|
| Preface | 21 |
| Product-Specific Documentation | 21 |
| How to Access TIBCO Documentation | 22 |
| How to Contact TIBCO Support | 22 |
| How to Join TIBCO Community | 22 |
| TDV SQL Support | 23 |
| Data Types | 23 |
| Summary of Data Types that TDV Supports | 23 |
| BOOLEAN | 26 |
| INTERVAL DAY | 27 |
| INTERVAL YEAR | 29 |
| XML | 30 |
| Subqueries in TDV | 31 |
| Scalar Subqueries | 31 |
| Correlated Subqueries | 31 |
| Consolidated List of TDV Keywords | 32 |
| Maximum SQL Length for Data Sources | 38 |
| TDV SQL Keywords and Syntax | 41 |
| BETWEEN | 42 |
| CREATE TABLE | 43 |
| CREATE TABLE AS SELECT | 43 |
| CROSS JOIN | 44 |
| DELETE | 44 |
| DISTINCT | 45 |
| DROP | 46 |
| EXCEPT | 46 |
| FULL OUTER JOIN | 47 |
| GROUP BY | 48 |
| HAVING | 49 |
| INNER JOIN | 49 |
| INSERT | 50 |
| INSERT, UPDATE, and DELETE on Views | 53 |
| INTERSECT | 53 |

| | |
|--|-----------|
| LEFT OUTER JOIN | 54 |
| OFFSET and FETCH | 55 |
| ORDER BY | 56 |
| PIVOT | 57 |
| UNPIVOT | 59 |
| RIGHT OUTER JOIN | 61 |
| SELECT | 62 |
| SELECT (Virtual Columns) | 63 |
| SEMIJOIN to a Procedure. | 64 |
| UNION. | 65 |
| UNION ALL. | 66 |
| UPDATE | 68 |
| WHERE. | 69 |
| WITH. | 69 |
| TDV Support for SQL Functions | 73 |
| About SQL Functions in TDV | 73 |
| TDV-Supported Analytical Functions | 74 |
| Window Clause | 76 |
| Default Assumptions. | 76 |
| RANGE and the Current Row. | 77 |
| RANGE as a Logical Offset | 77 |
| ROWS and the Current Row | 77 |
| ROWS and the Frame's Maximum Size. | 77 |
| AVG | 78 |
| CORR | 78 |
| COUNT | 78 |
| COVAR_POP | 79 |
| COVAR_SAMP | 79 |
| CUME_DIST | 80 |
| DENSE_RANK. | 80 |
| FIRST_VALUE. | 81 |
| LAG | 81 |
| LAST_VALUE | 82 |
| LEAD | 82 |
| LISTAGG | 82 |
| MAX. | 83 |
| MIN | 83 |
| NTH_VALUE | 84 |
| NTILE. | 84 |
| PERCENT_RANK | 84 |

| | |
|--|-----|
| PERCENTILE_CONT | 85 |
| PERCENTILE_DISC | 85 |
| RANK | 86 |
| RATIO_TO_REPORT | 86 |
| REGR_AVGX | 86 |
| REGR_AVGY | 87 |
| REGR_COUNT | 87 |
| REGR_INTERCEPT | 87 |
| REGR_R2 | 88 |
| REGR_SLOPE | 88 |
| REGR_SXX | 89 |
| REGR_SXY | 89 |
| REGR_SYY | 89 |
| ROW_NUMBER | 90 |
| STDDEV | 90 |
| STDDEV_POP | 91 |
| STDDEV_SAMP | 91 |
| SUM | 91 |
| VAR_POP | 92 |
| VAR_SAMP | 92 |
| VARIANCE | 92 |
| TDV-Supported Aggregate Functions | 93 |
| AVG | 95 |
| COUNT | 96 |
| DISTINCT in Aggregate Functions | 97 |
| MAX | 97 |
| MIN | 98 |
| SUM | 99 |
| XMLAGG | 100 |
| TDV-Supported Array SQL Script Functions | 101 |
| TDV-Supported Binary Functions | 101 |
| AND Functions | 102 |
| NOT Functions | 103 |
| OR Functions | 104 |
| SHL Functions | 105 |
| SHR Functions | 106 |
| XOR Functions | 108 |
| TDV-Supported Character Functions | 109 |
| ASCII | 112 |
| CHR | 112 |
| CONCAT | 113 |
| GET_JSON_OBJECT | 115 |
| INSTR | 116 |
| LENGTH | 117 |

| | |
|---|-----|
| LOWER | 118 |
| LPAD | 119 |
| PARTIAL_STRING_MASK | 120 |
| POSITION | 121 |
| REPLACE | 122 |
| RPAD | 123 |
| RTRIM | 124 |
| SPACE | 125 |
| SUBSTR and SUBSTRING | 126 |
| TRIM | 128 |
| UPPER | 129 |
| TDV-Supported Conditional Functions | 130 |
| COALESCE | 130 |
| DECODE | 131 |
| IFNULL | 132 |
| ISNULL | 132 |
| ISNUMERIC | 133 |
| NULLIF | 134 |
| NVL | 134 |
| NVL2 | 136 |
| TDV-Supported Convert Functions | 137 |
| CAST | 137 |
| FORMAT_DATE | 140 |
| PARSE_DATE | 142 |
| PARSE_TIME | 143 |
| PARSE_TIMESTAMP | 143 |
| TIMESTAMP | 144 |
| TO_BITSTRING | 144 |
| TO_CHAR | 144 |
| TO_DATE | 146 |
| TO_HEX | 146 |
| TO_NUMBER | 147 |
| TO_TIMESTAMP | 147 |
| TRUNC (for date/time) | 148 |
| TRUNC (for numbers) | 150 |
| TRUNCATE | 151 |
| TDV-Supported Cryptographic Functions | 151 |
| HASHMD2 | 151 |
| HASHMD4 | 152 |
| HASHSHA | 152 |
| HASHSHA1 | 153 |
| TDV-Supported Date Functions | 153 |
| CURRENT_DATE | 156 |
| CURRENT_TIME | 157 |

| | |
|---|-----|
| CURRENT_TIMESTAMP | 157 |
| DATEDIFF | 158 |
| DAY, MONTH, and YEAR | 159 |
| DAYS | 160 |
| DAYS_BETWEEN | 161 |
| DBTIMEZONE | 161 |
| EXTRACT | 162 |
| FROM_UNIXTIME | 163 |
| MONTHS_BETWEEN | 163 |
| NUMTODSINTERVAL | 164 |
| NUMTOYMINTERVAL | 164 |
| TZ_OFFSET | 165 |
| TZCONVERTOR | 165 |
| UTC_TO_TIMESTAMP | 166 |
| TDV-Supported JSON Functions | 167 |
| JSON_TABLE | 167 |
| Example 1: A Literal JSON Table | 170 |
| Example 2: Another Literal JSON Table, with Ignored Objects | 171 |
| Example 3: Retrieving Object Properties and Their Values | 173 |
| Example 4: JSON Content Provided by an External Table | 174 |
| Example 5: Subquery | 174 |
| Example 6: Conditional Logic with Key and Value Retrieval | 175 |
| Example 7: Invalid Keys and Values | 176 |
| Example 8: Nested Arrays | 177 |
| JSONPATH | 178 |
| TDV-Supported Numeric Functions | 179 |
| ABS | 181 |
| ACOS | 181 |
| ASIN | 182 |
| ATAN | 183 |
| ATAN2 | 183 |
| CEILING | 184 |
| COS | 185 |
| COSH | 185 |
| COT | 186 |
| DEGREES | 187 |
| EXP | 187 |
| FLOOR | 187 |
| LOG | 188 |
| PI | 188 |
| POWER | 189 |
| RADIANS | 189 |
| ROUND (for date/time) | 190 |
| ROUND (for numbers) | 193 |

| | |
|--|------------|
| SIN | 194 |
| SINH | 195 |
| SQRT | 195 |
| TAN | 196 |
| TANH | 197 |
| TDV-Supported Operator Functions | 197 |
| TDV-Supported Phonetic Functions | 198 |
| TDV-Supported Utility Function | 198 |
| TDV-Supported XML Functions | 199 |
| Identifier Escaping | 200 |
| Text Escaping | 201 |
| XMLATTRIBUTES | 201 |
| XMLCOMMENT | 202 |
| XMLCONCAT | 202 |
| XMLDOCUMENT | 203 |
| XMLELEMENT | 203 |
| XMLFOREST | 204 |
| XMLNAMESPACES | 205 |
| XMLPI | 205 |
| XMLQUERY | 205 |
| XMLTEXT | 207 |
| XPATH | 207 |
| XSLT | 208 |
| TDV Support for SQL Operators | 211 |
| Arithmetic Operators | 211 |
| Add | 212 |
| DECIMAL and NUMERIC Data Types | 212 |
| INTERVAL Type | 213 |
| Mixed Data Types | 214 |
| Concatenation | 217 |
| Divide | 217 |
| DECIMAL and NUMERIC Data Types | 218 |
| INTERVAL Type | 218 |
| Exponentiate | 219 |
| Factorial | 219 |
| Modulo | 219 |
| Multiply | 220 |
| DECIMAL and NUMERIC Data Types | 221 |
| INTERVAL Type | 221 |
| Mixed Data Types | 222 |
| Negate | 224 |
| INTERVAL Type | 224 |

| | |
|--|------------|
| Other Data Types | 224 |
| Subtract | 225 |
| DECIMAL and NUMERIC Data Types | 225 |
| INTERVAL Type | 226 |
| Mixed Data Types | 227 |
| Comparison Operators | 230 |
| Quantified Comparisons | 231 |
| Logical Operators | 232 |
| AND | 233 |
| NOT | 233 |
| OR | 234 |
| Condition Operators | 234 |
| CASE | 235 |
| Simple CASE | 235 |
| Searched CASE | 236 |
| COALESCE | 236 |
| DECODE | 237 |
| EXISTS and NOT EXISTS | 238 |
| IN and NOT IN | 239 |
| IS NOT NULL | 241 |
| IS NULL | 242 |
| LIKE | 242 |
| OVERLAPS | 243 |
| TDV Query Engine Options | 245 |
| DATA_SHIP_MODE Values | 246 |
| GROUP BY Options | 247 |
| INSERT, UPDATE, and DELETE Options | 249 |
| JOIN Options | 251 |
| DISABLE_PUSH (JOIN Option) | 252 |
| DISABLE_THREADS (JOIN Option) | 252 |
| FORCE_DISK (JOIN Option) | 253 |
| FORCE_ORDER (JOIN Option) | 253 |
| HASH (JOIN Option) | 254 |
| LEFT_CARDINALITY (JOIN Option) | 254 |
| NESTEDLOOP (JOIN Option) | 255 |
| PARTITION_SIZE (JOIN Option) | 255 |
| RIGHT_CARDINALITY (JOIN Option) | 256 |
| SEMIJOIN (JOIN Option) | 256 |
| SORTMERGE (JOIN Option) | 257 |
| SWAP_ORDER (JOIN Option) | 257 |
| ORDER BY Options | 258 |

| | |
|--|------------|
| DISABLE_PUSH (ORDER BY Option) | 258 |
| DISABLE_THREADS (ORDER BY Option) | 258 |
| FORCE_DISK (ORDER BY Option) | 259 |
| SELECT Options | 259 |
| CASE_SENSITIVE (SELECT Option) | 260 |
| DISABLE_CBO (SELECT Option) | 261 |
| DISABLE_DATA_CACHE (SELECT Option) | 261 |
| DISABLE_JOIN_PRUNER (SELECT Option) | 262 |
| DISABLE_PLAN_CACHE (SELECT Option) | 263 |
| DISABLE_PUSH (SELECT Option) | 263 |
| DISABLE_SELECTION_REWRITER (SELECT Option) | 263 |
| DISABLE_STATISTICS (SELECT Option) | 264 |
| DISABLE_THREADS (SELECT Option) | 264 |
| FORCE_DISK (SELECT Option) | 265 |
| IGNORE_TRAILING_SPACES (SELECT Option) | 265 |
| MAX_ROWS_LIMIT (SELECT Option) | 266 |
| ROWS_OFFSET (SELECT Option) | 268 |
| STRICT (SELECT Option) | 269 |
| PUSH_NULL_SELECTS (SELECT OPTION) | 270 |
| DISABLE_CONSTANT_FUNCTION_INLINING (SELECT OPTION) | 270 |
| DISABLE_UNION_PREAGGREGATOR (SELECT OPTION) | 270 |
| USE_COMPARABLE_ESTIMATES (SELECT OPTION) | 271 |
| UNION, INTERSECT, and EXCEPT Options | 271 |
| DISABLE_PUSH (UNION, INTERSECT, and EXCEPT Option) | 272 |
| FORCE_DISK (UNION, INTERSECT, and EXCEPT Option) | 272 |
| PARALLEL (UNION, INTERSECT, and EXCEPT Option) | 273 |
| ROUND_ROBIN (UNION, INTERSECT, and EXCEPT Option) | 273 |
| SORT_MERGE (UNION, INTERSECT, and EXCEPT Option) | 274 |
| TDV and Business Directory System Tables | 275 |
| Accessing TDV and Business Directory System Tables | 278 |
| ALL_BD_RESOURCES | 279 |
| ALL_CATALOGS | 280 |
| ALL_CATEGORIES | 280 |
| ALL_CATEGORY_VALUES | 281 |
| ALL_CLASSIFICATIONS | 281 |
| ALL_COLUMNS | 282 |
| ALL_COMMENTS | 284 |
| ALL_CUSTOM_PROPERTIES | 285 |
| ALL_CUSTOM_PROPERTY_CLASSIFICATIONS | 286 |
| ALL_CUSTOM_PROPERTY_GROUPS | 287 |

| | |
|--|-----|
| ALL_CUSTOM_PROPERTY_GROUPS_ASSOCIATIONS..... | 287 |
| ALL_DATASOURCES..... | 288 |
| ALL_DOMAINS..... | 289 |
| ALL_ENDPOINT_MAPPINGS..... | 289 |
| ALL_FOREIGN_KEYS..... | 290 |
| ALL_GROUPS..... | 293 |
| ALL_INDEXES..... | 293 |
| ALL_LINEAGE..... | 295 |
| ALL_PARAMETERS..... | 296 |
| ALL_PRINCIPAL_SET_MAPPINGS..... | 298 |
| ALL_PRIVILEGES..... | 299 |
| ALL_PROCEDURES..... | 300 |
| ALL_PUBLISHED_FOLDERS..... | 301 |
| ALL_RELATIONSHIP_COLUMNS..... | 302 |
| ALL_RELATIONSHIPS..... | 304 |
| ALL_RESOURCES..... | 307 |
| ALL_SCHEMAS..... | 308 |
| ALL_TABLES..... | 309 |
| ALL_USERS..... | 311 |
| ALL_USER_PROFILES..... | 311 |
| ALL_WATCHES..... | 312 |
| ALL_WSDL_OPERATIONS..... | 312 |
| DEPLOYMENT_PLAN_DETAIL_LOG..... | 314 |
| DEPLOYMENT_PLAN_LOG..... | 315 |
| DUAL..... | 316 |
| LOG_DISK..... | 317 |
| LOG_EVENTS..... | 317 |
| LOG_IO..... | 318 |
| LOG_MEMORY..... | 319 |
| SYS_CACHES..... | 319 |
| SYS_CLUSTER..... | 321 |
| SYS_DATA_OBJECTS..... | 322 |
| SYS_DATASOURCES..... | 322 |
| SYS_DEPLOYMENT_PLANS..... | 324 |
| SYS_PRINCIPAL_SETS..... | 325 |
| SYS_REQUESTS..... | 326 |

| | |
|---|------------|
| SYS_RESOURCE_SETS | 328 |
| SYS_SESSIONS | 329 |
| SYS_SITES | 331 |
| SYS_STATISTICS | 331 |
| SYS_TASKS | 333 |
| SYS_TRANSACTIONS | 335 |
| SYS_TRANSIENT_COLUMNS | 336 |
| SYS_TRANSIENT_SCHEMAS | 338 |
| SYS_TRANSIENT_TABLES | 338 |
| SYS_TRIGGERS | 340 |
| TEMPTABLE_LOG | 341 |
| TRANSACTION_LOG | 342 |
| USER_PROFILE | 344 |
| TDV SQL Script | 345 |
| SQL Script Overview | 345 |
| SQL Language Concepts | 346 |
| Identifiers | 347 |
| Data Types | 348 |
| Supported Data Types | 349 |
| Example (Declaring a Custom Data Type) | 351 |
| Example (Referencing a Custom Data Type) | 351 |
| Example (XML Data Type) | 352 |
| Value Expressions | 352 |
| Conditional Expressions | 353 |
| Literal Values | 354 |
| Noncursor Variables | 354 |
| Cursor Variables | 356 |
| Attributes of Cursors | 356 |
| Attributes of CURRENT_EXCEPTION | 358 |
| SQL Script Exceptions | 359 |
| SQL Script Keywords | 361 |
| SQL Script Procedures and Structure | 362 |
| Basic Structure of a SQL Script Procedure | 362 |
| SQL Script Procedure Header | 363 |
| PIPE Modifier | 364 |
| Compound Statements | 365 |
| Independent Transactions | 366 |
| Compensating Transactions | 368 |
| Exceptions | 369 |
| Raising and Handling Exceptions | 369 |

| | |
|--|-----|
| External Exceptions | 371 |
| SQL Script Statement Reference | 371 |
| BEGIN...END | 372 |
| CALL | 373 |
| CASE | 374 |
| CLOSE | 375 |
| COMMIT | 376 |
| CREATE TABLE | 376 |
| CREATE TABLE AS SELECT | 377 |
| CREATE INDEX | 377 |
| DECLARE Constants | 378 |
| DECLARE CURSOR of Type Variable | 378 |
| DECLARE <cursorName> CURSOR FOR | 379 |
| DECLARE EXCEPTION | 381 |
| DECLARE TYPE | 382 |
| DECLARE Variable | 383 |
| DECLARE VECTOR | 384 |
| DELETE | 390 |
| DROP TABLE | 391 |
| DROP INDEX | 391 |
| EXECUTE IMMEDIATE | 391 |
| FIND_INDEX | 392 |
| FETCH | 392 |
| FOR | 394 |
| IF | 395 |
| INSERT | 396 |
| ITERATE | 397 |
| LEAVE | 397 |
| LOOP | 398 |
| OPEN | 399 |
| PATH | 400 |
| RAISE | 401 |
| REPEAT | 402 |
| ROLLBACK | 402 |
| SELECT INTO | 403 |
| SET | 404 |
| TOP | 404 |
| UPDATE | 405 |
| WHILE | 406 |
| SQL Script Examples | 406 |
| Example 1 (Fetch All Rows) | 407 |
| Example 2 (Fetch All Categories) | 407 |
| Example 3 (User-Defined Type) | 408 |
| Example 4 (User-Defined Type) | 408 |

| | |
|---|------------|
| Example 5 (Pipe Variable) | 408 |
| Example 6 (Dynamic SQL Extract with Individual Inserts) | 408 |
| Example 7 (Dynamic SQL Inserts by Variable Name) | 409 |
| Example 8 (Prepackaged Query) | 409 |
| Example 9 (Exception Handling) | 409 |
| Example 10 (Row Declaration) | 410 |
| Example 11 (Avoiding Division-by-Zero Errors) | 410 |
| TDV Built-in Functions for XQuery | 411 |
| executeStatement | 411 |
| formatBooleanSequence | 412 |
| formatDateSequence | 412 |
| formatDecimalSequence | 413 |
| formatDoubleSequence | 413 |
| formatFloatSequence | 413 |
| formatIntegerSequence | 414 |
| formatStringSequence | 414 |
| formatTimeSequence | 415 |
| formatTimestampSequence | 415 |
| Java APIs for Custom Procedures | 417 |
| com.compositesw.extension | 417 |
| CustomCursor | 418 |
| CustomProcedure | 420 |
| CustomProcedureException | 423 |
| ExecutionEnvironment | 424 |
| ParameterInfo | 428 |
| ProcedureConstants | 432 |
| ProcedureReference | 435 |
| Data Type Mappings for Data Sources | 441 |
| TDV Data Source to JDBC Data Types | 442 |
| DataDirect Mainframe to TDV Data Types | 443 |
| DB2 to TDV Data Types | 444 |
| File - Cache to TDV Data Types | 446 |
| File - Delimited to TDV Data Types | 446 |
| Greenplum to TDV Data Types | 446 |
| HBase to TDV Data Types | 449 |

| | |
|---|-----|
| HSQLDB Database to TDV Data Types | 450 |
| Impala to TDV Data Types | 451 |
| Informix to TDV Data Types | 453 |
| LDAP to TDV Data Types | 454 |
| Microsoft Access to TDV Data Types | 455 |
| Microsoft Excel to TDV Data Types | 455 |
| Microsoft SQL Server to TDV Data Types | 456 |
| MySQL to TDV Data Types | 458 |
| Neoview to TDV Data Types | 460 |
| Netezza to TDV Data Types | 461 |
| OData to TDV Data Types | 463 |
| Oracle to TDV Data Types | 464 |
| Oracle NUMBER Data Types and TDV Data Types | 465 |
| Oracle to Data Types Common to All Versions | 466 |
| Oracle 9i to TDV Data Types | 467 |
| Oracle 10g to TDV Data Types | 468 |
| Oracle 11g to TDV Data Types | 469 |
| ParStream to TDV Data Types | 470 |
| PostgreSQL to TDV Data Types | 472 |
| Redshift Data Types | 475 |
| SAP HANA Data Types | 477 |
| Sybase ASE to TDV Data Types | 478 |
| Sybase IQ to TDV Data Types | 479 |
| Teradata to TDV Data Types | 481 |
| Vertica to TDV Data Types | 483 |
| Cache Data Type Mapping | 485 |
| DB2 Cache Mapping | 486 |
| DB2-on-z/OS Cache Mapping | 488 |
| File Cache Mapping | 489 |
| Greenplum Cache Mapping | 492 |
| HSQLDB Cache Mapping | 493 |
| Informix Cache Mapping | 495 |
| Microsoft Access Cache Mapping | 496 |
| Microsoft SQL Server Cache Mapping | 498 |
| MySQL Cache Mapping | 500 |
| Netezza Cache Mapping | 502 |
| Oracle Cache Mapping | 504 |
| PostgreSQL Cache Mapping | 505 |
| Redshift Cache Mapping | 507 |
| SAP HANA Cache Mapping | 509 |

| | |
|---|------------|
| Sybase ASE Cache Mapping | 511 |
| Sybase IQ Cache Mapping | 513 |
| Teradata Cache Mapping | 514 |
| Vertica Cache Mapping | 516 |
| Function Support for Data Sources | 519 |
| Pushing or Not Pushing Functions | 520 |
| Function Support Issues when Combining Data Sources | 520 |
| ASCII Function with Empty String Argument | 521 |
| Case Sensitivity and Trailing Spaces | 521 |
| Collating Sequence | 521 |
| Data Precision | 522 |
| Decimal Digit Limitation on Functions | 523 |
| INSTR Function | 523 |
| Interval Calculations | 523 |
| Mapping of Native to TDV Data Types Across TDV Versions | 524 |
| MERGE | 524 |
| MERGE and Data Sources | 525 |
| MERGE Examples | 526 |
| ORDER BY Clause | 530 |
| SPACE Function | 530 |
| SQL Server Sorting Order | 530 |
| Time Functions | 531 |
| Truncation vs. Rounding | 531 |
| TDV Native Function Support | 531 |
| TDV Aggregate Function Support | 532 |
| TDV Character Function Support | 532 |
| TDV Conditional Function Support | 533 |
| TDV Conversion Function Support | 533 |
| TDV Date Function Support | 533 |
| TDV Numeric Function Support | 534 |
| DataDirect Mainframe Function Support | 534 |
| DataDirect Mainframe Aggregate Function Support | 535 |
| DataDirect Mainframe Character Function Support | 535 |
| DataDirect Mainframe Conditional Function Support | 536 |
| DataDirect Mainframe Conversion Function Support | 536 |
| DataDirect Mainframe Date Function Support | 537 |
| DataDirect Mainframe Numeric Function Support | 537 |
| DataDirect Mainframe XML Function Support | 538 |
| DB2 Function Support | 539 |
| DB2 Aggregate Function Support | 540 |
| DB2 Analytic Function Support | 540 |
| DB2 Analytic Aggregate Function Support | 541 |

| | |
|---|-----|
| DB2 Character Function Support | 541 |
| DB2 Conditional Function Support | 542 |
| DB2 Conversion Function Support | 543 |
| DB2 Date Function Support | 543 |
| DB2 Linear Regression Function Support | 544 |
| DB2 Numeric Function Support | 545 |
| DB2 XML Function Support | 546 |
| DB2 Mainframe Function Support | 546 |
| DB2 Mainframe Aggregate Function Support | 547 |
| DB2 Mainframe Character Function Support | 547 |
| DB2 Mainframe Conditional Function Support | 548 |
| DB2 Mainframe Binary Function Support | 549 |
| DB2 Mainframe Conversion Function Support | 549 |
| DB2 Mainframe Date Function Support | 549 |
| DB2 Mainframe Numeric Function Support | 551 |
| DB2 Mainframe XML Function Support | 552 |
| File Function Support | 553 |
| File Aggregate Function Support | 553 |
| File Character Function Support | 553 |
| File Conversion Function Support | 554 |
| File Date Function Support | 554 |
| File Numeric Function Support | 555 |
| Greenplum Function Support | 556 |
| Greenplum Aggregate Function Support | 556 |
| Greenplum Analytic Function Support | 557 |
| Greenplum Analytic Aggregate Function Support | 558 |
| Greenplum Binary Function Support | 560 |
| Greenplum Character Function Support | 561 |
| Greenplum Conditional Function Support | 563 |
| Greenplum Conversion Function Support | 564 |
| Greenplum Date Function Support | 564 |
| Greenplum Numeric Function Support | 566 |
| Greenplum Time Function Support | 568 |
| HBase Function Support | 568 |
| HBase Aggregate Function Support | 569 |
| HBase Conversion Function Support | 569 |
| HBase Date Function Support | 569 |
| HBase Numeric Function Support | 570 |
| HBase String Function Support | 570 |
| HSQldb Function Support | 570 |
| HSQldb Aggregate Function Support | 571 |
| HSQldb Binary Function Support | 572 |
| HSQldb Conversion Function Support | 572 |
| HSQldb Date Function Support | 572 |

| | |
|--|-----|
| HSQLDB Numeric Function Support | 573 |
| HSQLDB String Function Support | 574 |
| Impala Function Support | 575 |
| Impala Aggregate Function Support | 576 |
| Impala Binary Function Support | 576 |
| Impala Conditional Function Support | 577 |
| Impala Conversion Function Support | 577 |
| Impala Date Function Support | 577 |
| Impala Numeric Function Support | 578 |
| Impala Push-Only Function Support | 580 |
| Impala String Function Support | 580 |
| Informix Function Support | 581 |
| Informix Aggregate Function Support | 582 |
| Informix Character Function Support | 582 |
| Informix Conditional Function Support | 583 |
| Informix Conversion Function Support | 583 |
| Informix Date Function Support | 584 |
| Informix Numeric Function Support | 584 |
| JDBC Function Support | 585 |
| JDBC Aggregate Function Support | 585 |
| JSON Function Support | 585 |
| Microsoft Access Function Support | 586 |
| Microsoft Access Aggregate Function Support | 586 |
| Microsoft Access Analytic Aggregate Function Support | 587 |
| Microsoft Access Character Function Support | 587 |
| Microsoft Access Conditional Function Support | 588 |
| Microsoft Access Conversion Function Support | 588 |
| Microsoft Access Date Function Support | 589 |
| Microsoft Access Numeric Function Support | 589 |
| Microsoft Excel Function Support | 590 |
| Microsoft SQL Server Function Support | 590 |
| Microsoft SQL Server Aggregate Function Support | 591 |
| Microsoft SQL Server Analytic Function Support | 591 |
| Microsoft SQL Server Analytic Aggregate Function Support | 592 |
| Microsoft SQL Server Character Function Support | 593 |
| Microsoft SQL Server Conditional Function Support | 594 |
| Microsoft SQL Server Conversion Function Support | 594 |
| Microsoft SQL Server Date Function Support | 595 |
| Microsoft SQL Server Encryption Function Support | 596 |
| Microsoft SQL Server Numeric Function Support | 596 |
| Microsoft SQL Server Time Function Support | 597 |
| MySQL Function Support | 597 |
| MySQL Aggregate Function Support | 598 |

| | |
|---|-----|
| MySQL Analytic Function Support | 598 |
| MySQL Analytic Aggregate Function Support | 598 |
| MySQL Character Function Support | 599 |
| MySQL Conditional Function Support | 599 |
| MySQL Conversion Function Support | 600 |
| MySQL Date Function Support | 601 |
| MySQL Numeric Function Support | 601 |
| MySQL Time Function Support | 602 |
| NeoView Function Support | 602 |
| NeoView Aggregate Function Support | 603 |
| NeoView Character Function Support | 603 |
| NeoView Conditional Function Support | 604 |
| NeoView Conversion Function Support | 604 |
| NeoView Date Function Support | 604 |
| NeoView Numeric Function Support | 605 |
| Netezza Function Support | 605 |
| Netezza Aggregate Function Support | 606 |
| Netezza Analytic Function Support | 608 |
| Netezza Analytic Aggregate Function Support | 609 |
| Netezza Binary Function Support | 610 |
| Netezza Character Function Support | 610 |
| Netezza Conditional Function Support | 612 |
| Netezza Conversion Function Support | 613 |
| Netezza Date Function Support | 613 |
| Netezza Numeric Function Support | 614 |
| Netezza Phonetic Function Support | 617 |
| Netezza Statistical Analytic Aggregate Function Support | 617 |
| Netezza Time Function Support | 618 |
| Oracle Function Support | 618 |
| Oracle Aggregate Function Support | 619 |
| Oracle Analytic Function Support | 620 |
| Oracle Analytic Aggregate Function Support | 621 |
| Oracle Binary Function Support | 622 |
| Oracle Character Function Support | 622 |
| Oracle Conditional Function Support | 623 |
| Oracle Conversion Function Support | 624 |
| Oracle Date Function Support | 625 |
| Oracle Encryption Function Support | 625 |
| Oracle Numeric Function Support | 626 |
| Oracle Time Function Support | 627 |
| Oracle XML Function Support | 627 |
| ParStream Function Support | 628 |
| PostgreSQL Function Support | 630 |
| PostgreSQL Aggregate Function Support | 631 |

| | |
|--|-----|
| PostgreSQL Analytic Aggregate Function Support | 631 |
| PostgreSQL Binary Function Support | 632 |
| PostgreSQL Character Function Support | 632 |
| PostgreSQL Conversion Function Support | 633 |
| PostgreSQL Date Function Support | 634 |
| PostgreSQL Numeric Function Support. | 634 |
| PostgreSQL Time Function Support | 636 |
| Redshift Function Support. | 636 |
| Redshift Aggregate Function Support | 636 |
| Redshift Analytical Function Support. | 637 |
| Redshift Character Function Support. | 637 |
| Redshift Conditional Function Support | 639 |
| Redshift Conversion Function Support | 639 |
| Redshift Date Function Support. | 640 |
| Redshift Numerical Function Support | 640 |
| Redshift Time Function Support | 641 |
| SAP HANA Function Support | 642 |
| SAP HANA Aggregate Function Support. | 642 |
| SAP HANA Analytical Function Support | 643 |
| SAP HANA Binary Function Support | 644 |
| SAP HANA Character Function Support | 645 |
| SAP HANA Conditional Function Support | 646 |
| SAP HANA Conversion Function Support | 646 |
| SAP HANA Date Function Support | 647 |
| SAP HANA Numeric Function Support | 648 |
| Sybase Function Support | 649 |
| Sybase Aggregate Function Support. | 650 |
| Sybase Character Function Support | 650 |
| Sybase Conditional Function Support | 651 |
| Sybase Conversion Function Support | 651 |
| Sybase Date Function Support | 651 |
| Sybase Numeric Function Support | 652 |
| Sybase ASE 15.7 MERGE Behavior | 653 |
| Sybase IQ Function Support. | 653 |
| Sybase IQ Aggregate Function Support | 654 |
| Sybase IQ Analytic Function Support | 655 |
| Sybase IQ Character Function Support. | 655 |
| Sybase IQ Conditional Function Support. | 656 |
| Sybase IQ Conversion Function Support. | 656 |
| Sybase IQ Date Function Support | 657 |
| Sybase IQ Numeric Function Support | 657 |
| Teradata Function Support. | 658 |
| Teradata Aggregate Function Support. | 659 |
| Teradata Analytic Function Support. | 659 |

| | |
|--|------------|
| Teradata Character Function Support | 660 |
| Teradata Conditional Function Support | 661 |
| Teradata Conversion Function Support | 662 |
| Teradata Date Function Support | 663 |
| Teradata Number Function Support | 663 |
| Vertica Function Support | 664 |
| Vertica Aggregate Function Support | 665 |
| Vertica Analytic Function Support | 667 |
| Vertica Binary Function Support | 668 |
| Vertica Character Function Support | 668 |
| Vertica Conditional Function Support | 671 |
| Vertica Conversion Function Support | 671 |
| Vertica Date Function Support | 672 |
| Vertica Numeric Function Support | 675 |
| Vertica OLAP Analytic Function Support | 676 |
| Vertica Time Series Function Support | 678 |
| XML Function Support | 678 |
| XML Aggregate Function Support | 679 |
| XML Character Function Support | 679 |
| XML Conversion Function Support | 680 |
| XML Date Function Support | 680 |
| XML Numeric Function Support | 680 |
| Custom Procedure Examples | 683 |
| About the Custom Procedure Examples Syntax | 683 |
| Example 1: Simple Query | 683 |
| Example 2: Simple Update | 686 |
| Example 3: External Update without Compensation | 689 |
| Example 4: Nontransactional External Update without Compensation | 694 |
| Example 5: Expression Evaluator | 698 |
| Example 6: Output Cursor | 702 |
| Example 7: Simple Procedure that Invokes Another Procedure | 706 |
| Function Support Summary | 711 |
| Time Zones | 737 |

Preface

Documentation for this and other TIBCO products is available on the TIBCO Documentation site. This site is updated more frequently than any documentation that might be included with the product. To ensure that you are accessing the latest available help topics, please visit:

- <https://docs.tibco.com>

Product-Specific Documentation

The following documents form the TIBCO® Data Virtualization(TDV) documentation set:

- *TIBCO TDV and Business Directory Release Notes* Read the release notes for a list of new and changed features. This document also contains lists of known issues and closed issues for this release.
- TDV Installation and Upgrade Guide
- TDV Administration Guide
- TDV Reference Guide
- TDV User Guide
- TDV Security Features Guide
- TDV Business Directory Guide
- TDV Application Programming Interface Guide
- TDV Tutorial Guide
- TDV Extensibility Guide
- TDV Getting Started Guide
- TDV Client Interfaces Guide
- TDV Adapter Guide
- TDV Discovery Guide
- TDV Active Cluster Guide
- TDV Monitor Guide
- TDV Northbay Example

How to Access TIBCO Documentation

Documentation for TIBCO products is available on the TIBCO Product Documentation website mainly in the HTML and PDF formats.

The TIBCO Product Documentation website is updated frequently and is more current than any other documentation included with the product. To access the latest documentation, visit <https://docs.tibco.com>.

Documentation for TIBCO Data Virtualization is available on <https://docs.tibco.com/products/tibco-data-virtualization-server>.

How to Contact TIBCO Support

You can contact TIBCO Support in the following ways:

- For an overview of TIBCO Support, visit <https://www.tibco.com/services/support>.
- For accessing the Support Knowledge Base and getting personalized content about products you are interested in, visit the TIBCO Support portal at <https://support.tibco.com>.
- For creating a Support case, you must have a valid maintenance or support contract with TIBCO. You also need a user name and password to log in to <https://support.tibco.com>. If you do not have a user name, you can request one by clicking **Register** on the website.

How to Join TIBCO Community

TIBCO Community is the official channel for TIBCO customers, partners, and employee subject matter experts to share and access their collective experience. TIBCO Community offers access to Q&A forums, product wikis, and best practices. It also offers access to extensions, adapters, solution accelerators, and tools that extend and enable customers to gain full value from TIBCO products. In addition, users can submit and vote on feature requests from within the [TIBCO Ideas Portal](https://community.tibco.com). For a free registration, go to <https://community.tibco.com>.

TDV SQL Support

TDV allows query specification and data updates using standard SQL. TDV supports a subset of ANSI SQL-92 and ANSI SQL-99.

- [Data Types, page 23](#)
- [Subqueries in TDV, page 31](#)
- [Consolidated List of TDV Keywords, page 32](#)

Data Types

This section summarizes the SQL data types that TDV supports, and provides detailed sections about data types with complex implementations.

- [Summary of Data Types that TDV Supports, page 23](#)
- [BOOLEAN, page 26](#)
- [INTERVAL DAY, page 27](#)
- [INTERVAL YEAR, page 29](#)
- [XML, page 30](#)

Summary of Data Types that TDV Supports

The following table discusses special considerations when using data types with TDV. Where more detailed discussion is required, separate sections are cross-referenced from the **Special Notes** column of the table.

| Data Types | Variants Supported | Special Notes |
|------------|----------------------|---|
| BINARY | BINARY, VARBINARY | <ul style="list-style-type: none"> • Behaves in a manner similar to STRING, but it is right-padded with zeroes rather than spaces. • Minimum length is 1. • Maximum length is 255. • BINARY or VARBINARY with length >255 is a BLOB. |

| Data Types | Variants Supported | Special Notes |
|------------|---|---|
| BIT | | |
| BLOB | BLOB | <ul style="list-style-type: none"> You can project (SELECT) BLOB columns. You can use BLOB only in the CAST function. |
| BOOLEAN | DATETIME | For more information, see BOOLEAN , page 26. |
| CLOB | CLOB | <ul style="list-style-type: none"> You can project (SELECT) CLOB columns. You can use CLOB only in the CAST function. |
| DATE | DATETIME | <ul style="list-style-type: none"> Month, day, year. |
| DECIMAL | DECIMAL, NUMERIC | <ul style="list-style-type: none"> Maximum precision is 23. An error is thrown if the number of digits to the left of the decimal point exceeds the precision specified for the type. For example, 12345.00 exceeds the limits of DECIMAL(4,2) and so throws an error. Minimum scale is -308; maximum scale is 308. Its scale (the digits to the right of the decimal point) is rounded if necessary to match the scale of the type designation. For example, 1.425 is rounded to 1.43 for DECIMAL(4,2). DECIMAL and NUMERIC data types are zero-padded on the right if the number of digits to the right of the decimal point is smaller than the scale of the type designation. For example, 1.425 becomes 1.42500 for DECIMAL(4,5). |
| DOUBLE | | |
| FLOAT | | |
| INTEGER | TINYINT, SMALLINT, INTEGER, BIGINT | <ul style="list-style-type: none"> A runtime error is thrown if a value is out of the valid range for the integer type. |

| Data Types | Variants Supported | Special Notes |
|----------------|--------------------|---|
| INTERVAL DAY | | <ul style="list-style-type: none"> Represents a duration of time. Intervals can be positive or negative. Not directly compatible with INTERVAL MONTH and INTERVAL YEAR. Can be used in arithmetic operations (addition, subtraction, division, and multiplication), and functions such as ABS, CAST, and EXTRACT. For more information, see INTERVAL DAY, page 27. |
| INTERVAL MONTH | | <ul style="list-style-type: none"> Represents a duration of time. Can be positive or negative. Not directly compatible with INTERVAL DAY and INTERVAL YEAR. Can be used in arithmetic operations (addition, subtraction, division, and multiplication), and functions such as ABS, CAST, and EXTRACT. |
| INTERVAL YEAR | | <ul style="list-style-type: none"> Represents a duration of time. Intervals can be positive or negative. Not directly compatible with INTERVAL DAY and INTERVAL MONTH. Can be used in arithmetic operations (addition, subtraction, division, and multiplication), and functions such as ABS, CAST, and EXTRACT. For more information, see INTERVAL YEAR, page 29. |
| LONGVARCHAR | | |
| REAL | | |

| Data Types | Variants Supported | Special Notes |
|------------|--------------------|--|
| STRING | CHAR, VARCHAR | <ul style="list-style-type: none">• Minimum length is 1.• If a CHAR is less than minimum length, it is right-padded with spaces.• Maximum length is 255.• CHAR or VARCHAR with length >255 is a CLOB.• Operations might pad a CHAR, even if it was not padded originally. So CONCAT (char10, char10) might return "A B " instead of "AB" as the result. |
| TIME | TIMESTAMP | <ul style="list-style-type: none">• Hours, minutes, seconds. |
| TIMESTAMP | | <ul style="list-style-type: none">• Month, day, year and hours, minutes, seconds.• Depending on formatting, may contain fractional seconds. |
| XML | | <ul style="list-style-type: none">• TDV support for the XML data type complies with the ANSI INCIT/ISO/IEC 9075 part 14 XML-related specifications.• For more information, see XML, page 30. |

BOOLEAN

As of TDV version 7.0.2, BOOLEAN complies with ANSI/ISO 2011 (draft), with the exceptions noted in the remarks below. Previous behavior is deprecated, although you can force the old behavior using a server configuration parameter, as described in [Overriding Standard-Compliant BOOLEAN Behavior](#), page 27.

- Character string literals “true” “false” and “unknown” can be CAST to BOOLEAN values TRUE, FALSE and UNKNOWN (NULL), respectively. The literal values are case-insensitive.
- Any other input values raise an error. Specifically, implicit conversion of non-zero numeric values to TRUE, and numeric values of zero to FALSE, raises an error.
- BOOLEAN types cannot be compared with other types without a CAST.
- Values of non-BOOLEAN types cannot be assigned to BOOLEAN targets directly. without a CAST. You must use a CASE to convert values of other types to TRUE, FALSE, or UNKNOWN, and then CAST those values to

BOOLEAN. For example, you cannot directly CAST(1 as BOOLEAN) to TRUE.

- Cannot Convert from BOOLEAN to non-BOOLEAN types or vice versa.
- BOOLEAN values cannot be function arguments. Specifically, the previous behavior of allowing BOOLEAN arguments to the following functions raises an error: CONCAT, DLE_DST, LE_DST, POSITION, REPEAT, TRIM, TS_FIRST_VALUE, and XMLTEXT.
- BOOLEAN types and values cannot be mixed with non-BOOLEAN types without a suitable CAST.
- Exception to the standard: TDV does not support {IS | IS NOT} {TRUE | FALSE | UNKNOWN} on BOOLEAN arguments.

Overriding Standard-Compliant BOOLEAN Behavior

You can use a configuration parameter to suppress the new, ANSI-compliant behavior and enable legacy BOOLEAN support. Legacy BOOLEAN support consists of mixing of BOOLEAN and non-BOOLEAN types without a CAST.

Legacy BOOLEAN support is deprecated as of TDV version 7.0.2.

The default value of this parameter is False.

To override standard-compliant BOOLEAN behavior

1. Select Administration > Configuration from the main Studio menu.
2. Navigate to Server > SQL Engine > SQL Language.
3. Set the parameter Allow Numeric Boolean Comparisons Assignments to True.

Changing the value has no effect until the next server restart.

INTERVAL DAY

INTERVAL DAY represents a duration of time that can be measured in days, hours, minutes, seconds, and fractions of seconds. INTERVAL can specify individual time units (for example, days only), pairs of time units (for example, days and hours), or mapping of units (for example, days to seconds). All INTERVAL DAY expressions are compatible with all other INTERVAL DAY expressions.

Syntax

```
INTERVAL 'dd hh:mm:ss.ff' DAY TO SECOND
INTERVAL 'dd hh:mm' DAY TO MINUTE
INTERVAL 'dd hh' DAY TO HOUR
```

INTERVAL 'dd' DAY
 INTERVAL 'hh' HOUR
 INTERVAL 'mm' MINUTE
 INTERVAL 'ss.ff' SECOND

Remarks

- In the format of date and time content:
 - A space separates the day value from the hour value.
 - A colon separates hour values from minute values, and minute values from seconds values.
 - A decimal point separates fractional seconds from seconds.
- For all time units, the default leading precision is 2. For example, the following pairs of expressions are equivalent:

INTERVAL '3' DAY
 INTERVAL '3' DAY(2)
 INTERVAL '3' MONTH
 INTERVAL '3' MONTH(2)

- For all time units, the maximum leading precision is 9 digits. An error is thrown if the number of digits to the left of the decimal point exceeds the leading precision.
- For seconds:
 - If only one precision value is specified, it designates fractional precision, which sets the maximum number of decimal places to the right of the decimal point.
 - If the fractional precision is exceeded, the extra digits are automatically truncated.
 - The default fractional precision for seconds is 6, so the following two expressions are equivalent:

INTERVAL '3' MINUTE(3) TO SECOND
 INTERVAL '3' MINUTE(3) to SECOND(6)

- The maximum fractional precision is 9 digits.
- To specify leading precision as well as fractional precision, enclose both in parentheses, separated by a comma:

INTERVAL '3.99' SECOND(2,6)

- Zero (0) is a valid fractional precision. For example, the following expression truncates fractional seconds to whole seconds:

INTERVAL '9:59' minutes to second(0)

- For details on using INTERVAL DAY in arithmetic operations and functions, see:
 - [Arithmetic Operators, page 211](#)
 - [CAST, page 137](#)
 - [EXTRACT, page 162](#)
 - [ABS, page 181](#)

INTERVAL YEAR

INTERVAL YEAR represents a unit of time that is measured in months and years. It can be expressed in years only, months only, or both year and months.

INTERVAL YEAR (which includes months) is not compatible with INTERVAL DAY, because a year can have 365 or 366 days, and a month can have 28, 29, 30, or 31 days.

Syntax

```
INTERVAL 'yy' YEAR [TO MONTH]
INTERVAL 'mm' MONTH
INTERVAL 'yy-mm' YEAR TO MONTH
```

Negative intervals can be represented in any of three formats:

```
-INTERVAL 'mm' MONTH
INTERVAL '-mm' MONTH
INTERVAL '-mm' MONTH
```

Remarks

- A dash separates the year and month values.
- In a year-month interval, the month value must not be greater than 11.
- The three formats for negative intervals can be intermixed. For example, the following resolves to an interval of -3 months:

```
-INTERVAL '-3' MONTH
```
- Default precision is 2. For example, the following expressions are equivalent:

```
INTERVAL '99' YEAR
INTERVAL '99' YEAR(2)
```
- The precision indicates the maximum number of digits in the leading number. For example, the expression below is invalid because its length exceeds the 2-digit precision in the year value.

```
INTERVAL '2001' YEAR(2)
```

- In a year-month interval, the precision applies only to the year:
INTERVAL '2001-09' YEAR(4) TO MONTH
- The maximum precision for years is 9 digits.
- For details on using INTERVAL YEAR in arithmetic operations and functions, see:
 - [Arithmetic Operators, page 211](#)
 - [CAST, page 137](#)
 - [EXTRACT, page 162](#)
 - [ABS, page 181](#)

XML

TDV support for the XML data type complies with the ANSI 9075 section 14 XML specification.

Syntax

```
XML [ ( { DOCUMENT | CONTENT | SEQUENCE }
[ ( ANY | UNTYPED | XMLSCHEMA schema-details ) ]
)]
```

Remarks

- schema-details is of the following form:
URI target-namespace-uri [LOCATION schema-location] [{ ELEMENT element-name | NAMESPACE namespace-uri [ELEMENT element-name] }]
[NO NAMESPACE [LOCATION schema-location] [{ ELEMENT element-name | NAMESPACE namespace-uri [ELEMENT element-name] }]
- target-namespace-uri, schema-location, and namespace-uri are STRING literals that represent valid URIs.
- element-name is any valid identifier.

Examples

```
CAST ('<item></item>' as XML (SEQUENCE))
CAST ('<entity></entity>' as XML (SEQUENCE(ANY)))
PROCEDURE item()
BEGIN
  DECLARE item XML (SEQUENCE(XMLSCHEMA URI 'http://www.w3.org/2001/XMLSchema-instance'
LOCATION 'http://www.w3.org/2001/XMLSchema-instance' ELEMENT xsi));
END
```

Subqueries in TDV

You can embed one SELECT statement within another SELECT statement. The embedded SQL statement is referred to as a subquery.

TDV supports using subqueries as values. See the section [EXISTS and NOT EXISTS](#), page 238.

Two types of subqueries are recognized: scalar subqueries and correlated subqueries.

Some subqueries reach row returned limitations before the query that you have written is complete. In cases where the data source allows a limit larger than 10,000 rows returned for subqueries, you can use the TDV In Clause Limit For SubQuery In Update And Delete configuration parameter to increase the subquery limit. There are many data source types that have limitations on the number of rows:

- returned from a subquery
- stored in memory
- stored in a cache

that cannot be modified. You must test your specific configuration and definitions to determine what is possible.

Scalar Subqueries

A scalar subquery is a subquery that returns a single value. It can be used anywhere a single column value or literal is valid.

A subquery can reside within a WHERE clause, a FROM clause, or a SELECT clause.

Example

```
SELECT *
FROM table1
WHERE column1 = (SELECT column1 FROM table2);
```

Correlated Subqueries

A correlated subquery is a subquery that contains a reference to a table that also appears in the outer query.

Syntax

```
SELECT outer_column
FROM outer_table
WHERE outer_column_value IN
  (SELECT inner_column FROM inner_table
WHERE inner_column = outer_column)
```

Remarks

- In the syntax above, outer_column is called the correlation variable, because it references the outer query from the inner query.
- A correlated subquery is used if a statement needs to process a table in the inner query for each row in the outer query.
- A correlated subquery cannot be evaluated independent of its outer query. The inner query is dependent on the data from the outer query.
- Correlated subqueries differ from simple queries because of their order of execution and the number of times they are executed. A correlated subquery is executed repeatedly, once for each candidate row selected by the outer query. It always refers to the table mentioned in the FROM clause of the outer query.

Example

The query lists the managers who are over 40 and who manage a sales person who is over quota and who does not work in the same sales office as the manager.

```
SELECT name
FROM salesreps mgrs
WHERE age > 40 AND mgrs.EMP_NO IN
(SELECT manager
FROM salesreps emps
WHERE emps.quota > emps.sales
AND emps.rep_office <> mgrs.rep_office)
```

Consolidated List of TDV Keywords

The following table is a consolidated list of TDV keywords; that is, character strings that have special meaning for the TDV parser. The table lists both reserved and nonreserved keywords.

Reserved Keywords

Reserved keywords are listed in bold font in the table.

- You cannot use reserved keywords as identifiers.

- Reserved keywords are not case-sensitive.
- If you want SQL statements to be portable across data sources, consult data source documentation for any additional reserved keywords they might have.

Nonreserved Keywords

Nonreserved keywords are listed in regular (nonbold) font in the table.

- It is advisable not to use nonreserved keywords as identifiers.
- If you choose to use a nonreserved keyword as an identifier, enclose it in double-quotes.
- Nonreserved keywords used as *identifiers* are case-sensitive; for example, “Absent” and “absent” are considered different identifiers.
- Nonreserved keywords used as *keywords* are not case-sensitive.

| TDV Parser Keywords | | | |
|---------------------|------------|----------------|---------------|
| ABSENT | ABSOLUTE | ACCORDING | ACTION |
| ADD | ALL | ALLOCATE | ALTER |
| AND | ANY | ARE | AS |
| ASC | ASSERTION | AT | AUTHORIZATION |
| AVG | BASE64 | BEGIN | BETWEEN |
| BINARY | BIT | BIT_LENGTH | BOOLEAN |
| BOTH | BREADTH | BY | CALL |
| CASCADE | CASCADEED | CASE | CAST |
| CATALOG | CHAR | CHAR_LENGTH | CHARACTER |
| CHARACTER_LENGTH | CHECK | CLOSE | COALESCE |
| COLLATE | COLLATION | COLLECTION | COLUMN |
| COLUMNS | COMMIT | CONNECT | CONNECTION |
| CONSTANT | CONSTRAINT | CONSTRAINTS | CONTENT |
| CONTINUE | CONVERT | CORRESPONDING | COUNT |

| TDV Parser Keywords | | | |
|---------------------|-------------------|---------------|--------------|
| CREATE | CROSS | CURRENT | CURRENT_DATE |
| CURRENT_TIME | CURRENT_TIMESTAMP | CURRENT_USER | CURSOR |
| CYCLE | D | DATE | DAY |
| DAYS | DEALLOCATE | DEC | DECIMAL |
| DECLARE | DEFAULT | DEFERRABLE | DEFERRED |
| DELETE | DENSE_RANK | DEPTH | DESC |
| DESCRIBE | DESCRIPTOR | DIAGNOSTICS | DISCONNECT |
| DISTINCT | DO | DOCUMENT | DOMAIN |
| DOUBLE | DOW | DOY | DROP |
| ELEMENT | ELSE | ELSEIF | EMPTY |
| END | END-EXEC | EPOCH | ESCAPE |
| EXCEPT | EXCEPTION | EXCLUDE | EXEC |
| EXECUTE | EXISTS | EXPLAIN | EXTERNAL |
| EXTRACT | FALSE | FETCH | FIRST |
| FLOAT | FN | FOLLOWING | FOR |
| FOREIGN | FROM | FULL | GET |
| GLOBAL | GO | GOTO | GRANT |
| GROUP | HAVING | HEX | HOUR |
| HOURS | ID | IDENTITY | IF |
| IGNORE | IMMEDIATE | IN | INDEPENDENT |
| INDEX | INDICATOR | INITIALLY | INNER |
| INOUT | INPUT | INSENSITIVE | INSERT |
| INT | INTEGER | INTERSECT | INTERVAL |

| TDV Parser Keywords | | | |
|---------------------|--------------|--------------|--------------|
| INTO | IS | ISOLATION | ITERATE |
| JOIN | KEEP | KEY | LANGUAGE |
| LAST | LATEST | LEADING | LEAVE |
| LEFT | LEVEL | LIKE | LOCAL |
| LOCATION | LONGVARCHAR | LOOP | LOWER |
| MATCH | MAX | MICROSECOND | MICROSECONDS |
| MILLISECOND | MILLISECONDS | MIN | MINUTE |
| MINUTES | MODULE | MONTH | MONTHS |
| NAME | NAMES | NAMESPACE | NATIONAL |
| NATURAL | NCHAR | NEXT | NIL |
| NO | NOT | NULL | NULLIF |
| NULLS | NUMERIC | OCTET_LENGTH | OF |
| OFFSET | OJ | ON | ONLY |
| OPEN | OPTION | OR | ORDER |
| OTHERS | OUT | OUTER | OUTPUT |
| OVER | OVERLAPS | PAD | PARTIAL |
| PARTITION | PASSING | PATH | PIPE |
| POSITION | PRECEDING | PRECISION | PREPARE |
| PRESERVE | PRIMARY | PRIOR | PRIVILEGES |
| PROCEDURE | PUBLIC | QUARTER | RAISE |
| RANGE | READ | REAL | RECURSIVE |
| REF | REFERENCES | RELATIVE | REPEAT |
| REPLACE | RESTRICT | RETURNING | REVOKE |

| TDV Parser Keywords | | | |
|-----------------------------|-----------------------------|----------------------------|-------------------------------|
| RIGHT | ROLLBACK | ROW | ROWS |
| SCHEMA | SCROLL | SEARCH | SECOND |
| SECONDS | SECTION | SELECT | SEQUENCE |
| SESSION | SESSION_USER | SET | SIZE |
| SMALLINT | SOME | SOURCE | SPACE |
| SQL | SQL_BIGINT | SQL_BINARY | SQL_BIT |
| SQL_CHAR | SQL_DATE | SQL_DECIMAL | SQL_DOUBLE |
| SQL_FLOAT | SQL_GUID | SQL_INTEGER | SQL_INTERVAL_DAY |
| SQL_INTERVAL_DAY_TO_HOUR | SQL_INTERVAL_DAY_TO_MINUTE | SQL_INTERVAL_DAY_TO_SECOND | SQL_INTERVAL_HOUR |
| SQL_INTERVAL_HOUR_TO_MINUTE | SQL_INTERVAL_HOUR_TO_SECOND | SQL_INTERVAL_MINUTE | SQL_INTERVAL_MINUTE_TO_SECOND |
| SQL_INTERVAL_MONTH | SQL_INTERVAL_SECOND | SQL_INTERVAL_YEAR | SQL_INTERVAL_YEAR_TO_MONTH |
| SQL_LONGVARIABLE | SQL_LONGVARIABLE | SQL_NUMERIC | SQL_REAL |
| SQL_SMALLINT | SQL_TIME | SQL_TIMESTAMP | SQL_TINYINT |
| SQL_TSI_DAY | SQL_TSI_FRAC_SECOND | SQL_TSI_HOUR | SQL_TSI_MINUTE |
| SQL_TSI_MONTH | SQL_TSI_QUARTER | SQL_TSI_SECOND | SQL_TSI_WEEK |
| SQL_TSI_YEAR | SQL_VARBINARY | SQL_VARCHAR | SQL_WCHAR |
| SQL_WLONGVARIABLE | SQL_WVARIABLE | SQLCODE | SQLERROR |
| SQLSTATE | STRIP | SUBSTRING | SUM |

| TDV Parser Keywords | | | |
|---------------------|---------------|---------------|---------------|
| SYSTEM_USER | T | TABLE | TEMPORARY |
| THEN | TIES | TIME | TIMESERIES |
| TIMESTAMP | TIMESTAMPADD | TIMESTAMPDIFF | TIMEZONE_HOUR |
| TIMEZONE_MIN UTE | TO | TOP | TRAILING |
| TRANSACTION | TRANSLATE | TRANSLATION | TRIM |
| TRUE | TS | TYPE | UNBOUNDED |
| UNION | UNIQUE | UNKNOWN | UNTIL |
| UNTYPED | UPDATE | UPPER | URI |
| USAGE | USE | USER | USING |
| VALUE | VALUES | VARBINARY | VARCHAR |
| VARYING | VECTOR | VIEW | WEEK |
| WHEN | WHENEVER | WHERE | WHILE |
| WHITESPACE | WITH | WITHIN | WORK |
| WRITE | XML | XMLAGG | XMLATTRIBUTES |
| XMLBINARY | XMLCAST | XMLCOMMENT | XMLCONCAT |
| XMLDOCUMENT T | XMLELEMENT | XMLEXISTS | XMLFOREST |
| XMLITERATE | XMLNAMESPACES | XMLPARSE | XMLPI |
| XMLQUERY | XMLSCHEMA | XMLSERIALIZE | XMLTABLE |
| XMLTEXT | XMLVALIDATE | YEAR | YEARS |
| ZONE | | | |

Maximum SQL Length for Data Sources

The maximum SQL command lengths for each data source in different versions of TDV are as follows.

| Data Source Type | Maximum SQL Length Prior to 6.2 SP4 | Maximum SQL Length, 6.2 SP4 and Later |
|----------------------|-------------------------------------|---------------------------------------|
| TDV | 16000 | unchanged |
| DataDirect Mainframe | 1000 | unchanged |
| Greenplum | 4000 | 65536 |
| Hive, Hive2 | 8000 | 32768 |
| IBM DB2 | 8000 | unchanged |
| IBM DB2 Type 2 | 8000 | 131072 |
| IBM DB2 Mainframe | 2097152 | unchanged |
| Informix | 1024 | 65536 |
| JDBC | 1024 | unchanged |
| Microsoft Access | 1000 | 32768 |
| Microsoft Excel | 1024 | unchanged |
| MySQL | 4000 | 65536 |
| Netezza | 4000 (v3.0: 1024) | 65536 |
| Oracle 9i | 64000 | unchanged |
| Oracle 10g, 11g | 64000 | 131072 |
| Oracle Type 2 | 64000 | unchanged |
| PostgreSQL | 32768 | 65536 |
| REST | 1024 | unchanged |
| SOAP | 1024 | unchanged |
| SQL Server | 8000 | 32768 |

| Data Source Type | Maximum SQL Length Prior to 6.2 SP4 | Maximum SQL Length, 6.2 SP4 and Later |
|-------------------|--|--|
| Sybase, Sybase IQ | 4000 | 65536 |
| Sybase IQ Type 2 | 4000 | unchanged |
| Vertica | 32768 | 65536 |
| Web Services | 1024 | unchanged |
| XMLFILE | 16000 | unchanged |
| XMLHTTP | 1024 | unchanged |

TDV SQL Keywords and Syntax

This topic describes the syntax for the SQL keywords supported by TDV:

- [BETWEEN](#), page 42
- [CREATE TABLE](#), page 43
- [CREATE TABLE AS SELECT](#), page 43
- [CROSS JOIN](#), page 44
- [DELETE](#), page 44
- [DISTINCT](#), page 45
- [DROP](#), page 46
- [EXCEPT](#), page 46
- [FULL OUTER JOIN](#), page 47
- [GROUP BY](#), page 48
- [HAVING](#), page 49
- [INNER JOIN](#), page 49
- [INSERT](#), page 50
- [INSERT, UPDATE, and DELETE on Views](#), page 53
- [INTERSECT](#), page 53
- [LEFT OUTER JOIN](#), page 54
- [OFFSET and FETCH](#), page 55
- [ORDER BY](#), page 56
- [PIVOT](#), page 57
- [UNPIVOT](#), page 59
- [RIGHT OUTER JOIN](#), page 61
- [SELECT](#), page 62
- [SELECT \(Virtual Columns\)](#), page 63
- [SEMIJOIN to a Procedure](#), page 64
- [UNION](#), page 65
- [UNION ALL](#), page 66

- [UPDATE](#), page 68
- [WHERE](#), page 69
- [WITH](#), page 69

BETWEEN

BETWEEN is a filter that chooses values within a specified range. When used with the optional keyword NOT, BETWEEN chooses values outside of a specified range.

Syntax

[NOT] BETWEEN low_value AND high_value

Remarks

- The BETWEEN range contains a low value and a high value. The low value must be less than or equal to the high value.
- Both low and high values are included in the search.
- BETWEEN can be used in both WHERE and HAVING clauses.
- BETWEEN works with character strings, numbers, and date-times. Only the values that are identical to the search values are returned.
- BETWEEN is equivalent to using `<=` and `>=` with this syntax:
`WHERE test_column >= low_value AND test_column <= high_value`

Example (Between Values)

```
SELECT ProductID, ProductName
FROM /shared/examples/ds_orders/products
WHERE UnitPrice BETWEEN 50 and 100
```

This query returns the product ID and name for all products whose unit price is between 50 and 100, inclusive.

Example (Between Dates)

```
SELECT OrderID
FROM /shared/examples/ds_orders/orders
WHERE OrderDate BETWEEN DATE '2012-05-03' AND DATE '2012-05-04'
```

This query returns the order ID for all orders with an order date of May 3 or May 4, 2012.

CREATE TABLE

Creates a new table in the database.

Syntax

```
CREATE TABLE table_name (
    column1 datatype,
    column2 datatype,
    column3 datatype,
    ....
);
```

CREATE TABLE AS SELECT

Create a table from an existing table by copying the existing table's columns. The new table is populated with the records from the existing table.

Creates a TEMPORARY table as a copy of an existing table.

Syntax

```
CREATE [TEMPORARY] TABLE table-name AS QUERY_EXPRESSION
```

```
CREATE [TEMPORARY] TABLE new_table
AS (SELECT * FROM old_table);
```

Remarks

- The QUERY_EXPRESSION can be any select query without an ORDER BY or LIMIT clause.
- The temporary table will be empty on first access, can optionally be returned to empty state at every COMMIT by using the ON COMMIT clause. The temporary tables are automatically cleaned up by the server at the end of the user session. You can also explicitly drop them if needed in between the session.
- If most of the queries are going against a particular database, the performance of the joins on temporary table with the persisted table might be better with a specific temporary table storage location. The privileges associated with the Temporary Table Container affect the user who can create and use temporary tables if the DDL Container is set. The temporary table storage location can be changed by editing the Temporary Table Container configuration parameter through Studio.

Examples

```
CREATE TABLE queenbee
AS (SELECT * FROM babybee);
```

OR

```
CREATE TEMPORARY TABLE queenbee
AS (SELECT * FROM babybee);
```

CROSS JOIN

CROSS JOIN takes the Cartesian product—that is, all combinations of each table in the join.

Syntax

```
table1 CROSS JOIN table2
```

Example

```
SELECT *
FROM city CROSS JOIN attraction;
```

If city has 4 rows and attraction has 5 rows, CROSS JOIN returns 20 rows.

DELETE

TDV supports the regular SQL DELETE statement.

See also [INSERT, UPDATE, and DELETE on Views, page 53](#).

Syntax

```
DELETE FROM <table>
[WHERE <criteria>]
```

Remarks

- The WHERE clause can have a subquery.
- All database objects referenced in the subquery must be from the same data source as the target of the DELETE.
- IN subqueries can be scalar or not.

- Depending on the relational operator, quantified subqueries may need to be scalar.
- If the subquery references incorrect rows, unexpected target rows might be affected.
- If the underlying data source has the truncate_table capability set, then the hints use_truncate and try_truncate can be used with the DELETE keyword.

Example (Deleting All Rows)

The following example deletes all the rows in the orders table:

```
DELETE FROM /shared/examples/ds_orders/orders
```

Example (Deleting Specific Rows)

The following example deletes the row where the product ID is 44 in the orders table:

```
DELETE FROM /shared/examples/ds_orders/orders
WHERE ProductID = 44
```

Example (Using a Subquery)

The following example uses a subquery:

```
DELETE FROM /shared/examples/ds_orders/orders
WHERE ProductID IN (SELECT ProductID FROM /shared/examples/ds_orders2/orderdetails)
```

Example (Using hints for Truncate)

The following example uses a subquery:

```
DELETE {option use_truncate} FROM /shared/examples/ds_orders/orders
```

In this case, the query engine will run TRUNCATE TABLE, if the truncate capability is set for the data source in the capabilities file. If not, an error will be displayed.

```
DELETE {option try_truncate} FROM /shared/examples/ds_orders/orders
```

In this case, the query engine will run TRUNCATE TABLE, if the truncate capability is set for the data source in the capabilities file. If not, DELETE statement will be executed.

DISTINCT

DISTINCT eliminates duplicate rows from the result set.

Syntax

```
DISTINCT columnX
```

Remarks

- If any column has a NULL value, it is treated like any other value.
- If you have DISTINCT and GROUP BY in the SELECT clause, the GROUP BY is applied first before DISTINCT.
- DISTINCT supports all data types, including: BLOB, CLOB, and XML.
- DISTINCT in the SELECT clause and DISTINCT in an aggregate function do not return the same result.

Example

```
SELECT DISTINCT StateOrProvince
FROM /shared/examples/ds_orders/customers customers
```

DROP

Removes a table definition and all the data, indexes, triggers, constraints and permission specifications for that table.

Syntax

```
DROP TABLE [IF EXISTS] table_name;
```

Remarks

- DROP TABLE throws an error if the table does not exist, or if other database objects depend on it.
- DROP TABLE IF EXISTS does not throw an error if the table does not exist. It throws an error if other database objects depend on the table.

EXCEPT

EXCEPT is like the UNION statement, except that EXCEPT produces rows that result from the first query but not the second.

Note: EXCEPT is known as MINUS in Oracle.

Syntax

```
<query_expression>
```

```
EXCEPT [ALL]
<query_expression>
```

Remarks

- Unlike UNION and INTERSECT, EXCEPT is not commutative. That is, A EXCEPT B is not the same as B EXCEPT A. Otherwise, the rules are the same as for UNION.
- When you use EXCEPT ALL, if a row appears x times in the first table and y times in the second table, it appears z times in the result table, where z is x - y or 0 (zero), whichever is greater.
- EXCEPT is similar to EXCEPT ALL and eliminates the duplicates.
- Using only EXCEPT provides results that have no duplicates in their result set.
- Using EXCEPT ALL includes rows that have duplicate values.

Example (EXCEPT)

The following query on a file in the Studio resource tree lists the cities where suppliers live but no customers live.

```
SELECT City
FROM /shared/examples/ds_inventory/suppliers
EXCEPT
SELECT City
FROM /shared/examples/ds_orders/customers
```

Oakland is the only city in the supplier's result set that is not in the customers result set.

Example (EXCEPT ALL)

```
SELECT City
FROM /shared/examples/ds_inventory/suppliers
EXCEPT ALL
SELECT City
FROM /shared/examples/ds_orders/customers
```

Adding ALL returns rows that have duplicates in the suppliers result set.

FULL OUTER JOIN

FULL OUTER JOIN merges two streams of incoming rows and produces one stream containing the SQL FULL OUTER JOIN of both streams.

Syntax

```
Select *
FROM table1
FULL OUTER JOIN table2
ON table1.column_name = table2.column_name;
```

Remarks

- The FULL OUTER JOIN combines the results of both left and right outer joins.
- When no matching rows exist for rows on the left side of the JOIN key word, NULL values are returned from the result set on the right.
- When no matching rows exist for rows on the right side of the JOIN key word, NULL values are returned from the result set on the left.
- The query engine hashes the lesser side and streams the greater side over it.

Example

```
SELECT *
FROM /shared/examples/ds_orders/orderdetails orderdetails
FULL OUTER JOIN /shared/examples/ds_orders/products products
ON orderdetails.ProductID = products.ProductID;
```

GROUP BY

GROUP BY is used when multiple columns from one or more tables are selected and at least one aggregate function appears in the SELECT statement. In that case, you need to GROUP BY all the selected columns except the ones operated on by the aggregate function.

All data types (including: BLOB, CLOB, and XML) are supported by GROUP BY.

Syntax

```
SELECT column1, ... column_n, aggregate_function (expression)
FROM table
GROUP BY column1, ... column_n;
```

Example (GROUP BY with Multiple Inner Joins)

```
SELECT orderdetails.Status, count (orderdetails.Status) as Item_Count
FROM /shared/examples/ds_orders/orderdetails Orderdetails
INNER JOIN /shared/examples/ds_inventory/products Products
ON orderdetails.ProductID = products.ProductID
INNER JOIN /shared/examples/ds_orders/orders Orders
ON orders.OrderID = orderdetails.OrderID
GROUP BY orderdetails.Status
```

Example (GROUP BY with Columns Specified by Ordinal Position)

Columns that are to be used for grouping can be defined by the integer that represents the ordinal position in which the SELECT occurred. If all columns of a table are selected (SELECT *), you can use the column position in the table (expressed as an integer).

```
SELECT ProductId, UnitsSold, UnitPrice
FROM /shared/examples/ds_inventory/inventorytransactions InventoryTransactions
GROUP BY 2 DESC, 1, 3
```

This sample query selects the three columns ProductId, UnitsSold, and UnitPrice from the inventorytransactions table and groups the results first by UnitsSold (in descending order), then by ProductId (in ascending order), and then by UnitPrice (in ascending order).

HAVING

The HAVING clause is used in combination with GROUP BY. You can use HAVING in a SELECT statement to filter the records that a GROUP BY returns.

Syntax

```
GROUP BY column1, ... column_n
HAVING condition1 ... condition_n;
```

Example

```
SELECT OrderID, SUM (orderdetails.Quantity) sumQuantity
FROM /shared/examples/ds_orders/orderdetails
GROUP BY OrderID
HAVING SUM (orderdetails.Quantity) > 10
```

The example has 50 unique OrderID values. SUM (orderdetails.Quantity) returns 296, but adding the GROUP BY clause causes the results to have a separate SUM (quantity) value. HAVING SUM adds a filter to that result set.

INNER JOIN

INNER JOIN return rows when there is at least one match in both tables.

Syntax

```
SELECT columnA, ... columnX
FROM table1
```

```
INNER JOIN table2
ON table1.columnA = table2.columnA
```

Example

```
SELECT products.ProductName, products.ProductID
FROM /shared/examples/ds_inventory/products products
INNER JOIN /shared/examples/ds_inventory/products products_1
ON products.ProductID = products_1.ProductID
```

INSERT

The INSERT statement adds rows to a table. You can insert a single row or multiple rows with one statement.

You can use an INSERT statement only in a SQL script or from a JDBC/ODBC call. See also [INSERT, UPDATE, and DELETE on Views, page 53](#).

The INSERT INTO statement can also be used to insert a complete row of values without specifying the column names. Values must be specified for every column in the table, in the order specified by the DDL. If the number of values is not the same as the number of columns in the table, or if a value is not allowed for a particular data type, an exception is thrown.

The INSERT statement itself does not return a result, but the database system returns a message indicating how many rows have been affected. You can then verify the insertion by querying the data source.

Warning: If a network connection is dropped while data is being moved through TDV using INSERT statements, queries are likely to fail. The TDV Server cannot reconcile the data when the connection is re-established. You will need to determine when the failure occurred, how much data might have moved, and the best way to resolve the failure.

TDV supports INSERT only for the following data sources.

| | |
|--|--------------|
| • TDV | • Oracle |
| • DataDirect—Mainframe | • PostgreSQL |
| • File—Delimited | • REST |
| • Informix | • SOAP |
| • Microsoft Access (Windows platform only) | • Sybase ASE |

- | | |
|------------------------|-------------|
| • Microsoft Excel | • Sybase IQ |
| • Microsoft SQL Server | • Teradata |
| • MySQL | • Vertica |
| • Netezza | |

Note: For add-ons such as adapters, consult the documentation to find out if INSERT is supported.

Three forms of INSERT syntax are supported for TDV as a data source.

Syntax 1

```
INSERT INTO <table_name> DEFAULT VALUES
```

Syntax 2

```
INSERT INTO <table_name> [(<columnA, ... columnX>)]
VALUES (<valueList>)[,(<valueList>)]*
```

Syntax 3

```
INSERT INTO <table_name> [(<columnA, ... columnX>)]
<queryExpression>
```

Opening and closing parentheses are used for grouping; <queryExpression> indicates a SELECT statement.

Listing of the columns is optional. In all cases, the number and type of the values must be equal and consistent with the number of columns in the row or as specified. See [Example \(Multi-Row INSERT with <queryExpression>\)](#), page 52.

Remarks

- The system automatically discards any ORDER BY in the subqueries, because it is not useful to sort the subquery.
- In a multi-row INSERT, the query result must contain the same number of columns in the same order as the column list in the INSERT statement, and the data types must be compatible, column by column.
- If a non-nullable column is set to NULL, the data source throws a runtime exception.
- INSERT statements should include all non-nullable columns.
- Derived columns cannot be present in an INSERT statement.

Example (Single-Row INSERT)

```

PROCEDURE sc2()
BEGIN
  INSERT INTO
    /shared/examples/ds_inventory/products (ProductID, ProductName, UnitPrice)
  VALUES (23, 'monitor', 500.00);
END

```

Example (Multi-Row INSERT)

```

PROCEDURE sc2()
BEGIN
  INSERT INTO
    /shared/examples/ds_inventory/products (ProductID, ProductName,
      UnitPrice)
  VALUES
    (41, 'monitor', 1000/10 * 1),
    (42, 'monitor', 1000/10 * 1),
    (43, 'monitor', 1000/10 * 1);
END

```

Example (Multi-Row INSERT with <queryExpression>)

```

PROCEDURE get_open_orders(OUT numOpen INTEGER)
BEGIN
  -- Clear the table
  DELETE FROM /users/composite/test/sources/mysql/updates;

  -- Get all open orders
  INSERT INTO /users/composite/test/sources/mysql/updates
    (c_bigint, c_vchar)
  SELECT OrderID, Status
  FROM /shared/tutorial/sources/ds_orders/orderdetails
  WHERE Status = 'Open';

  -- Return number of open orders
  SELECT count(*) INTO numOpen
  FROM /users/composite/test/sources/mysql/updates;
END

```

Example (INSERT with DEFAULT)

```

INSERT INTO Customers (FirstName, LastName, Country)
VALUES ('joe', 'Ely', DEFAULT)

```

An exception is thrown if the target database does not support the DEFAULT keyword.

A runtime exception is thrown if the column does not have a default defined and is non-nullable.

Example (INSERT with DEFAULT VALUES)

INSERT INTO Customers DEFAULT VALUES

If a DEFAULT VALUES clause is specified, a single row is inserted into a table containing the appropriate defaults (possibly null) in every column. It is an error if any column has no default.

INSERT, UPDATE, and DELETE on Views

INSERT, UPDATE, and DELETE on views are supported as defined by SQL standards, under the following conditions:

- A view is updatable only if:
 - It is defined to be a direct row and column subset of some base table, or a direct row and column subset of some other updatable view.
 - The SQL of the view does not include DISTINCT, GROUP BY, or HAVING.
 - The FROM clause of the view refers to exactly one table reference, and that table reference identifies either a base table or an updatable view.
- Derived columns are not updatable.
- A view with an aggregate expression in projection is not updatable whether GROUP BY is present or not.

INTERSECT

INTERSECT returns only rows that appear in both queries. The rules are the same as those listed for [UNION, page 65](#).

Syntax

```
<query_expression>
INTERSECT [ALL]
<query_expression>
```

Remarks

- According to SQL standards, INTERSECT takes precedence over UNION and EXCEPT.

- With INTERSECT ALL, if a row appears x times in the first table and y times in the second table, the row appears z times in the result table, where z is the lesser of x and y.
- INTERSECT is similar to INTERSECT ALL, plus INTERSECT eliminates duplicate rows.

Example (INTERSECT)

The following query lists the cities where suppliers and customers are found, and eliminates duplicate rows.

```
SELECT City
FROM /shared/examples/ds_inventory/suppliers
INTERSECT
SELECT City
FROM /shared/examples/ds_orders/customers
```

Example (INTERSECT ALL)

The following query lists the cities where suppliers and customers are found, but does not eliminate duplicate rows.

```
SELECT City
FROM /shared/examples/ds_inventory/suppliers
INTERSECT ALL
SELECT City
FROM /shared/examples/ds_orders/customers
```

LEFT OUTER JOIN

LEFT OUTER JOIN returns all records of the left table even if the join-condition does not find any matching record in the right table.

Remarks

- A left outer join (or left join) closely resembles a right outer join, except with the treatment of the tables reversed.
- Every row from the left table appears in the joined table at least once.
- If no matching row from the right table exists, NULL appears in columns from the right table for those records that have no match in the left table.
- A left outer join returns all the values from the left table and matched values from the right table (NULL in case of no matching join predicate).
- The query engine hashes the lesser side and streams the greater side over it.

Syntax

```
SELECT columns
FROM tableA
LEFT OUTER JOIN tableB
ON tableA.columnX = tableB.columnX
```

Example

```
SELECT *
FROM /shared/examples/ds_orders/products products
LEFT OUTER JOIN /shared/examples/ds_orders/orderdetails orderdetails
ON products.ProductID = orderdetails.ProductID
```

OFFSET and FETCH

When a table is sorted (preferably using ORDER BY on a primary key), OFFSET can be used to skip a specified number of rows. OFFSET is usually combined with FETCH NEXT value ROWS ONLY to support pagination, selecting a specific subset of rows in a table sorted on a primary key.

Note: For a discussion of how this option, MAX_ROWS_LIMIT, OFFSET, FETCH and the maxRows JDBC/ODBC parameter work together, see [MAX_ROWS_LIMIT \(SELECT Option\), page 266](#).

Syntax

```
SELECT *
FROM /table_path/table_name
ORDER BY column_name_PK
OFFSET value1 ROWS FETCH NEXT value2 ROWS ONLY
```

In the syntax, column_name_PK is a primary key that ensures consistent table ordering, value1 is the number of rows to skip, and value2 is the number of rows to fetch from the source.

Remarks

It is recommended that OFFSET be used with ORDER BY on a primary key to ensure repeatability for display of reliable subsets for paginated display of desired rows. The sorting with ORDER BY can be performed on any column, but if the table is changing rapidly, the ordering cannot be guaranteed. Tables that change in a more predictable manner might be safe to sort on any column with acceptably consistent output.

This function only applies to the top-level SELECT, and the result set from a query specifying OFFSET and FETCH is executed independently of other invocations.

Note: OFFSET and FETCH should not be used in a TDV view.

Example

```
SELECT orderdetails.OrderDetailID,
orderdetails.OrderID,
    orderdetails.ProductID,
    orderdetails.Status,
FROM /shared/examples/ds_orders/orderdetails
ORDER BY OrderDetailID
OFFSET 10 ROWS FETCH NEXT 10 ROWS ONLY
```

In this example, OrderDetailID is a primary key, and the OFFSET line tells the query engine to skip the first 10 rows and return the next 10.

ORDER BY

This function sorts columns in ascending order (the default) or descending order (if specified, as shown in the example below).

Syntax

```
ORDER BY columnA [ASC | DESC] [NULLS FIRST | NULLS LAST] [, columnB [ASC | DESC]
[NULLS FIRST | NULLS LAST], ... ]]
```

Remarks

- If you do not specify ORDER BY, the order is undefined. Without ORDER BY, the sort order can be different with two runs of the same SQL query.
- When you specify multiple columns, the results are sorted by the first column specified, then by the second column within the first column, and so on.
- By default, the TDV Server returns NULLs first for ASC and NULLs last for DESC.
 - Microsoft, Sybase, SQL Server, MySQL and Informix data sources also use these default values.
 - Oracle and DB2 data sources use opposite defaults.
- TDV supports ORDER BY in analytical functions as well as SELECT clauses.

Note: Oracle and Netezza also support ORDER BY in analytical functions. Microsoft data sources do not.

Example (ORDER BY without a Function)

```
SELECT *
FROM /shared/examples/ds_inventory/inventorytransactions InventoryTransactions
ORDER BY ProductID, UnitsSold DESC
```

This example selects all columns from the inventorytransactions table, sorts them by ProductID (in ascending order), and within each ProductID sorts them by UnitsSold (in descending order).

Example (ORDER BY with Columns Specified by Ordinal Position)

The order that the columns are selected can be replaced by the integer that represents the ordinal position where the SELECT occurred. If all columns of a table are selected by SELECT *, the column position in the table (expressed as an integer) can be used.

```
SELECT ProductId, UnitsSold, UnitPrice
FROM /shared/examples/ds_inventory/inventorytransactions InventoryTransactions
ORDER BY 2 DESC, 1
```

This example selects the three columns ProductId, UnitsSold, and UnitPrice from the inventorytransactions table, and orders the results first by UnitsSold, in descending order, and then by ProductId, in ascending order.

Example (ORDER BY with a Multiplication Function)

```
SELECT ProductId, UnitsSold * UnitPrice
FROM /shared/examples/ds_inventory/inventorytransactions
ORDER BY ProductID, UnitsSold * UnitPrice DESC
```

This example selects ProductId, UnitsSold, and UnitPrice from inventorytransactions and sorts them by ProductID in ascending order, and within each ProductID sorts them in descending order of the results obtained by multiplying UnitsSold by UnitPrice.

PIVOT

PIVOT operator rotates a table-valued expression by turning the unique values from one column in the expression into multiple columns in the output, and performs aggregations where they are required on any remaining column values that are wanted in the final output.

Syntax

```

pivot_clause : table_reference
              PIVOT LEFT_PAREN aggregate_function ( AS alias )? ( COMMA aggregate_function ( AS alias )? ) *
                pivot_for_clause
                pivot_in_clause
              RIGHT_PAREN

pivot_for_clause : FOR ( column
                      | LEFT_PAREN column ( COMMA column ) * RIGHT_PAREN
                    )

pivot_in_clause : IN LEFT_PAREN ( expression ( AS identifier )? ( COMMA expression ( AS identifier )? ) *
                                | pivot_multiple_columns ( COMMA pivot_multiple_columns ) *
                                | subquery
                                | ANY
                              )
                RIGHT_PAREN

pivot_multiple_columns : LEFT_PAREN expression ( COMMA expression ) * RIGHT_PAREN
                       ( AS identifier )?

```

Remarks

- The pivot operator will take the left side table_reference's projections as inputs. The argument to the aggregate_function must be a projection from the table_reference.
- The column specified in the pivot_for_clause clause must be a projection from table_reference. And will be matched against the expressions in the IN clause.
- All other projections in the table_referenced will be GROUP'ed BY.

Example

```

SELECT VendorID, Emp1, Emp2, Emp3, Emp4, Emp4
FROM
(SELECT PurchaseOrderID, EmployeeID, VendorID
FROM Purchasing.PurchaseOrderHeader) p
PIVOT
(COUNT (PurchaseOrderID)
FOR EmployeeID IN
( 250 as Emp1, 251 as Emp2, 256 as Emp3, 257 as Emp4, 260 as Emp5 )
) AS pvt

```

The PIVOT operator essentially invokes the following SQL

```

select VendorID, COUNT (PurchaseOrderID), EmployeeID
FROM Purchasing.PurchaseOrderHeader
WHERE EmployeeID IN 250, 251, 256, 257, 260)
GROUP BY VendorID, EmployeeID

```

An example result set of the above SQL is:

```
PIVOT
(
COUNT (PurchaseOrderID)
FOR EmployeeID IN
( 250 as Emp1, 251 as Emp2, 256 as Emp3, 257 as Emp4, 260 as Emp5)
)
```

| VendorID | Emp1 | Emp2 | Emp3 | Emp4 | Emp5 |
|----------|------|------|------|------|------|
| 1492 | 2 | 5 | 4 | 4 | 4 |
| 1494 | 2 | 5 | 4 | 5 | 4 |
| 1496 | 2 | 4 | 4 | 5 | 5 |
| 1498 | 2 | 5 | 4 | 4 | 4 |
| 1500 | 3 | 4 | 4 | 5 | 4 |

UNPIVOT

The UNPIVOT operator takes a table expression (table, procedure, or JOIN) and rotates columns into rows.

Syntax

```
unpivot_clause : table_reference UNPIVOT ( ( INCLUDE | EXCLUDE ) NULLS )?
                LEFT_PAREN ( identifier | LEFT_PAREN identifier ( COMMA identifier )+ RIGHT_PAREN )
                unpivot_for_clause
                unpivot_in_clause
                RIGHT_PAREN (AS)? identifier
```

```
unpivot_for_clause : FOR identifier
```

```
unpivot_in_clause : IN LEFT_PAREN ( column ( AS string_constant )? ( COMMA column ( AS
string_constant )? )*
                | unpivot_multiple_columns ( COMMA unpivot_multiple_columns )*
                )
                RIGHT_PAREN
```

```
unpivot_multiple_columns : LEFT_PAREN column ( COMMA column )* RIGHT_PAREN
                ( AS string_constant )?
```

Remarks

- The table expression can be a table, procedure, or JOIN.
- The result of the table expression will be fed into the UNPIVOT operator

Example for Projections

The UNPIVOT operator introduces new projections specified by the identifiers immediately following the UNPIVOT and FOR keyword

```
UNPIVOT (LabelOldColumnValues .... FOR LabeOldColumnNames
```

LabelOldColumnValues and LabeOldColumnNames will become the two new columns. LabeOldColumnNames will contain the names of the unpivoted columns. LabelOldColumnValues will contain the unpivoted column's values.

```
UNPIVOT (LabelOldColumnValues FOR LabeOldColumnNames IN (columnA, columnB)
```

Example for Renaming Columns

Old column names can be renamed by specifying the new name as a string constant in the IN clause.

In the example below, instead of the strings 'columnA' and 'columnB', we will see the strings 'rename1' and 'rename2'

```
UNPIVOT ... FOR LabeOldColumnNames IN (columnA as 'rename1', columnB as 'rename2')
O LabeOldColumnNames LabelOldColumnValues
```

```
1 rename1 a1
1 rename2 a2
2 rename1 b1
2 rename2 b2
3 rename1 c1
3 rename2 c2
```

Example for Multiple Column Sets

```
UNPIVOT ( (LabelOldColumnValues1, LabelOldColumnValues2, LabelOldColumnValues3) FOR
LabeOldColumnNames IN ( (columnA, columnB, columnC), (columnD, columnE, columnF) )
```

```
O columnA columnB columnC columnD columnE columnF
1 a1 b1 c1 d1 e1 f1
2 a2 b2 c2 d2 e2 f2
3 b3 c3 d3 e3 f3
```

will be rotated to

```
O LabeOldColumnNames LabelOldColumnValues1 LabelOldColumnValues2
LabelOldColumnValues3
```

```
-----
1 columnA_columnB_columnC a1 b1 c1
1 columnD_columnE_columnF d1 e1 f1
2 columnA_columnB_columnC a2 b2 c2
2 columnD_columnE_columnF d2 e2 f2
3 columnA_columnB_columnC a3 b3 c3
3 columnD_columnE_columnF d3 e3 f3
```


Example for Renaming Multiple Column Sets

```
UNPIVOT ( (LabelOldColumnValues1, LabelOldColumnValues2, LabelOldColumnValues3) FOR
LabelOldColumnNames IN ( (columnA, columnB, columnC) as 'gold', (columnD, columnE, columnF)
as 'silver'))
```

```
O LabelOldColumnNames LabelOldColumnValues1 LabelOldColumnValues2
LabelOldColumnValues3
```

```
1 gold a1 b1 c1
1 silver d1 e1 f1
2 gold a2 b2 c2
2 silver d2 e2 f2
3 gold a3 b3 c3
3 silver d3 e3 f3
```

RIGHT OUTER JOIN

RIGHT OUTER JOIN returns all records of the right table even if the join-condition does not find any matching record in the left table.

Syntax

```
SELECT columns
FROM tableA
RIGHT OUTER JOIN tableB
ON tableA.columnX = tableB.columnX
```

Remarks

- A right outer join (or right join) closely resembles a left outer join, except with the treatment of the tables reversed.
- Every row from the right table appears in the joined table at least once.
- If no matching row from the left table exists, NULL appears in columns from the left table for those records that have no match in the right table.
- A right outer join returns all the values from the right table and matched values from the left table (NULL in case of no matching join predicate).
- The query engine hashes the lesser side and streams the greater side over it.

Example

```
SELECT *
FROM /shared/examples/ds_orders/products products
RIGHT OUTER JOIN /shared/examples/ds_orders/orderdetails orderdetails
ON products.ProductID = orderdetails.ProductID
```

SELECT

The SELECT statement selects rows from a table.

Syntax

TDV supports the SELECT statement in various forms:

- With a FROM clause and a table
- With a FROM clause and a system table named DUAL for queries that do not require a table of actual data
- Without a FROM clause
- With the syntax `SELECT <expression> [,<expression>];` for example:
`SELECT 2+2`

Remarks

- If a network connection is dropped while data is being moved through the TDV Server using SELECT statements, queries are likely to fail. The TDV Server cannot reconcile the data when the connection is re-established. You will need to determine when the failure occurred, how much data might have moved, and the best way to resolve the failure.

Overriding SELECT Option Behavior

You can use a configuration parameter to revert the TDV Server default behavior for how SELECTs propagate between the parent and child. The SELECT in TDV will behave in the following manner unless the old SELECT option compatibility mode is enabled:

- Joining views that have conflicting select options results in an exception.
- Selecting options in joined tables are merged.
- Select options in derived tables, scalar subqueries, quantified comparisons will not affect its parent query

To revert the SELECT option behavior

1. Select Administration > Configuration from the main Studio menu.
2. Locate the Enable Old Select Option Compatibility Mode configuration parameter.
3. Set the parameter to True.

- Changing the value has no effect until the next server restart.

SELECT (Virtual Columns)

Besides supporting standard SQL SELECT statements, TDV supports the definition of “virtual columns” in the projection list for a view. After virtual columns are declared, you can use them in a query anywhere that you can use a literal.

The primary use of a virtual column is in procedures included in the FROM clause of a query. However, you can also use virtual columns in WHERE, HAVING, and JOIN ON clauses. Including them in the GROUP BY and ORDER BY clauses is acceptable, but it has no effect (like literals).

Syntax

```
{DECLARE columnName columnType [DEFAULT literalValue]}
```

The virtual column is declared in the SELECT clause, as follows:

```
SELECT c1, {DECLARE columnNameA columnTypeA,  
c2, {DECLARE columnNameB columnTypeB DEFAULT xx} ...
```

Remarks

- Virtual columns are unqualified, so their names must be unique and different from the names of items in the FROM clause.

For example, if you select FROM a table with a column named ColumnOne, the virtual column should not be named ColumnOne.

- When a query using virtual columns is executed, the query engine analyzes the predicates (such as a WHERE clause) to look for columnName = literal expressions. These clauses are removed from the query and the literal is replaced, much like a ? (question mark) is replaced in a prepared statement.

For example, the following statement

```
SELECT * FROM V1 WHERE columnName = 99
```

would become

```
SELECT T1.column1, 99, T1.column2  
FROM /some/table T1, Procedure1 (5,99) P1, Procedure2 (concat(99,'abc')) P2  
WHERE (99 > T1.column1) AND (T1.someKey = P2.someKey)
```

- The use of `columnName = literal` is important. Other types of comparison operators do not result in setting the value. The literal can be a single literal or an expression containing only functions and literals, like `concat('abc','def')`.
- Relationship optimization applies to virtual columns. This means that if the query has `columnName = otherColumn` and there is a predicate for `otherColumn = 5`, the query engine figures out that `columnName = 5` is also true and set that for you.
- It is possible when using outer joins for the WHERE clause to be illegally applied to the inner side of the join. When this happens, the query engine is unable to do the replacement, resulting in an error message that may or may not be easy to understand.
- If no DEFAULT value is specified for a virtual column, the column's value must be specified in the WHERE clause; otherwise, an error occurs.
- If a DEFAULT value is specified, it is used if no WHERE clause setting is found.
- If a virtual column is set to more than one value, you get an error.

Example

The following SELECT statement defines view V1:

```
SELECT T1.column1, {DECLARE columnName INTEGER DEFAULT 50}, T1.column2
FROM /some/table T1, Procedure1 (5, columnName) P1, Procedure2 (concat(columnName,'abc')) P2
WHERE (columnName > T1.column1) AND (T1.someKey = P2.someKey)
```

SEMIJOIN to a Procedure

A SEMIJOIN to a procedure is the logical equivalent of a semijoin to a table.

Syntax

```
<table_expression>
[LEFT OUTER | RIGHT OUTER | INNER | FULL OUTER] PROCEDURE JOIN
<procedure> ProcedureAlias
ON <condition_expression>
```

This syntax conveys that for each unique-value set of procedure inputs, the procedure on the right is called once. The results from each call are combined and treated as a row that is fed into the join. The join operates like a nonprocedure-join of the same type.

Remarks

- The special syntax given here always has a procedure on the right side and allows you to deviate from the normal rule that a procedure's input parameters must be literal expressions.
- When using this syntax, the procedure's input parameters can include references to any item from the table expression on the left, and only from that context. That is, only values from inside the left-side subquery can be used. The values from other scopes cannot be used.
- All the input value combinations are tracked and are not repeated to call the procedure again.
- Regarding using the PROCEDURE keyword:
 - Without the PROCEDURE keyword, your procedure is called exactly once.
 - With the keyword, your procedure is called zero or more times, depending on the left side of the join.

Example

```
(T1 LEFT OUTER JOIN T2 ON T1.x = T2.x)
INNER PROCEDURE JOIN
MyProc(T1.y+T2.y) P1 ON (T1.z = P1.z)
```

UNION

UNION works like [UNION ALL, page 66](#), except that it does not produce duplicate rows.

Syntax

```
<query_expression>
UNION
<query_expression>
```

Remarks

- The SELECT clause lists in the two queries must have the same number of projections.
- Corresponding columns in the two queries must be listed in the same order.
- Corresponding columns must have the same data type or must be implicitly convertible to the same data type.

- An ORDER BY clause can appear in only the final query of the UNION statement. The sort is applied to the final combined result.
- GROUP BY and HAVING can be specified in the individual queries only. They cannot be used to affect the final result.
- For the purposes of a SET operation, two NULLs are duplicates of each other.

Example

The following sample query lists the states where authors and publishers are located in the `authors` table and `publishers` table, respectively.

```
SELECT state FROM authors
UNION
SELECT state FROM publishers
```

UNION ALL

UNION ALL combines two tables, row by row. Implement UNION ALL by using the **SQL** panel of Studio Modeler.

Syntax

```
SELECT columnA [, columnB, ... ]
FROM table1
UNION ALL
SELECT columnA [, columnB, ... ]
FROM table2
```

Remarks

Multiple column selections can be made, but the number of columns and the column data types should match. All queries in a SQL statement containing the UNION ALL function must have an equal number of expressions in their target lists, as shown in the following example.

Example

```
SELECT ProductID, ProductName, UnitPrice
FROM /shared/examples/ds_inventory/products products
UNION ALL
SELECT ProductID, ProductName, UnitPrice
FROM /shared/examples/ds_inventory/products products_1
```

Example (To Contrast with Results of UNION)

Suppose that table T1 has columns C1, C2, and C3, and table T2 has columns Ca, Cb, Cc.

Table T1 has these values.

| | | |
|-----|-------|---------|
| | | |
| 001 | Hello | Goodbye |
| 002 | Hola | Adios |
| 003 | Aloha | Aloha |

Table T2 has these values.

| | | |
|-----|-------|-------------|
| | | |
| 003 | Aloha | Aloha |
| 004 | Alo | Adieu |
| 007 | Ciao | Arrivederci |

You execute the following query:

```
SELECT C1 C2 C3 FROM T1
UNION ALL
SELECT Ci Cii Ciii FROM T2
```

The results returned are shown in the table below.

| | | |
|-----|-------|-------------|
| 001 | Hello | Goodbye |
| 002 | Hola | Adios |
| 003 | Aloha | Aloha |
| 003 | Aloha | Aloha |
| 004 | Alo | Adieu |
| 007 | Ciao | Arrivederci |

This result set from UNION ALL contrasts with the output of the UNION function, which omits the repeated value of 003.

UPDATE

You can update a physical table view based on a single physical table. See [INSERT, UPDATE, and DELETE on Views, page 53](#) for rules on updating views.

Syntax

```
UPDATE <table>
SET <column> = <expression> [, <column> = <expression>]*
[WHERE <criteria>]
```

Remarks

- If a non-nullable column is set to NULL, the data source layer throws a runtime exception.
- If the column is set to an invalid value, the data source layer throws a runtime exception.
- The WHERE clause can have a subquery.
 - All database objects referenced in the subquery must be from the same data source as the target of the UPDATE.
 - IN subqueries can be scalar or not.
 - Depending on the relational operator, quantified subqueries may need to be scalar.
 - If the subquery references incorrect rows, unexpected target rows might be affected.
- The SET clause can have a subquery.
 - All database objects referenced in the subquery must be from the same data source as the target of the UPDATE.
 - Subqueries of SET clauses must be scalar (that is, return one value as one row).

Example (Using UPDATE with SET)

```
PROCEDURE sc5()
BEGIN
  UPDATE
    /shared/examples/ds_inventory/products
  SET
    ProductName = 'Apple';
END
```


Example (Using UPDATE with SET and WHERE)

```

PROCEDURE sc6()
BEGIN
  UPDATE
    /shared/examples/ds_inventory/products
  SET
    ProductName = 'Lexington Z24'
  WHERE
    ProductID = 5;
END

```

Example (Using UPDATE with SET and a Subquery)

```

PROCEDURE sc8()
BEGIN
  UPDATE /shared/examples/ds_orders2/products
  SET
    ProductName = 'abc'
  WHERE
    ProductID IN
    (SELECT ProductID FROM /shared/examples/ds_orders2/orderdetails);
END

```

WHERE

The WHERE clause extracts only those records that meet some criterion.

Syntax

```

SELECT columnA [, columnB, ... ]
FROM tableX
WHERE columnY <expression>

```

Example

```

SELECT ProductID, ProductName, ProductDescription
FROM /shared/examples/ds_inventory/products Products
WHERE ReorderLevel > 5

```

WITH

A WITH clause, used at the beginning of a SQL query, defines aggregations that in turn can be referred to in the main query and in other WITH statements as if they were physical tables.

A WITH statement can be used to create a common table expression (CTE). A CTE can be thought of as a temporary result set that is defined within the execution scope of a single SELECT, INSERT, UPDATE, DELETE, or CREATE VIEW statement. A CTE is not stored as an object, and persists only for the duration of the query.

Syntax

```
WITH queryName AS (query expression)
[ , ...]
mainQueryExpression
```

Remarks

- A WITH clause can also refer to a sibling WITH definition (second example below).
- You can first name a query expression and use it within the main query expression by referring to it. If an expression occurs more than once or is complex, moving it out provides clarity.
- The WITH query is run once and the results are stored in the equivalent of a temporary table, which is scanned whenever the results are used. For certain types of queries, this scanning can reduce the burden on the data source.

Example

Suppose that you have a Web service that returns employee data with the following columns:

- employeeNo (the employee's number)
- employeeName (the employee's name)
- manager (the employee number of the employee's manager)

The following query lists all the employees with the details on their respective managers:

```
WITH us_employees AS
(SELECT employeeNo, employeeName, manager FROM employee_webservice WHERE country =
'US')
SELECT e.employeeNo, e.employeeName, 'works for', e.manager,
'who is', m.employeeNo, m.employeeName
FROM us_employees e, us_employees m
WHERE e.manager = m.employeeNo
```

The advantage of using WITH in this scenario is that it invokes the Web service only once, which in turn enhances query execution performance.

Example (Two WITH Clauses that Do Not Refer to Each Other)

In the following example, X and Y are unique names that do not refer to each other (that is, the value of X is not the same as the value of Y).

WITH

```
X as (SELECT * From Foo),
```

```
Y as (SELECT * From X)
```

```
Select * From Y
```

Example (WITH Statement for Common Table Expressions)

The following example shows the components of the CTE structure: expression name, column list, and query.

```
WITH Sales_CTE (PersonID, OrderID, Year)
```

```
AS
```

```
-- Define the CTE query.
```

```
(
```

```
    SELECT PersonID, OrderID, OYEAR(OrderDate) AS Year
```

```
    FROM Sales.OrderHeader
```

```
    WHERE PersonID IS NOT NULL
```

```
)
```

```
-- Define the outer query referencing the CTE name.
```

```
SELECT PersonID, COUNT(OrderID) AS Total, Year
```

```
FROM Sales_CTE
```

```
GROUP BY Year, PersonID
```

```
ORDER BY PersonID, Year
```


TDV Support for SQL Functions

TDV supports SQL functions that manipulate alphabetical, numeric, date, time, and XML data types.

This topic provides usage, syntax, and examples for the SQL functions supported in TDV. After a brief introduction, the functions are presented in groups by type:

- [About SQL Functions in TDV, page 73](#)
- [TDV-Supported Analytical Functions, page 74](#)
- [TDV-Supported Aggregate Functions, page 93](#)
- [TDV-Supported Array SQL Script Functions, page 101](#)
- [TDV-Supported Binary Functions, page 101](#)
- [TDV-Supported Character Functions, page 109](#)
- [TDV-Supported Conditional Functions, page 130](#)
- [TDV-Supported Convert Functions, page 137](#)
- [TDV-Supported Cryptographic Functions, page 151](#)
- [TDV-Supported Date Functions, page 153](#)
- [TDV-Supported JSON Functions, page 167](#)
- [TDV-Supported Numeric Functions, page 179](#)
- [TDV-Supported Operator Functions, page 197](#)
- [TDV-Supported Phonetic Functions, page 198](#)
- [TDV-Supported Utility Function, page 198](#)
- [TDV-Supported XML Functions, page 199](#)

About SQL Functions in TDV

When you design a query in the Model panel of the view editor in the Studio Modeler, the SQL of the query is automatically generated and displayed in the SQL panel for the view. You can also use the SQL panel to type SQL statements directly.

Note: Do not use keywords (function names, operator names, and so on) as the names of TDV resources.

In DECIMAL and NUMERIC arguments, *p* refers to the precision (the combined maximum number of digits that can be stored to the left and the right of the decimal point) and *s* refers to the scale (the maximum number of digits that can be stored to the right of the decimal point). Scale can be specified only if precision is specified.

TDV-Supported Analytical Functions

Analytical functions produce summaries, reports, and statistics on large amounts of static data. TDV supports more than three dozen such functions.

Analytical functions are OLAP (on-line analytic processing) functions that operate on large amounts of static data. Most SQL functions are OLTP (on-line transaction processing) functions that operate as quickly as possible on discrete amounts of dynamic, transactional data.

Analytical functions are generally characterized by an OVER keyword and a window clause. (See [Window Clause](#), page 76.)

Limitation

- Large data sets can be very slow when using analytical functions.
- Teradata does not support the RANGE keyword. It only supports the ROWS keyword.
- For analytical functions that support the windowing clause, TDV does not push to Teradata without you explicitly supplying the windowing clause. Teradata implicitly adds ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING for analytical functions that do not supply a windowing clause. In TDV and ANSI SQL, RANGE UNBOUNDED PRECEDING is supplied.
- Teradata (version 16) does not support the RANGE keyword.

TDV supports the following analytical functions:

- [AVG](#), page 78
- [CORR](#), page 78
- [COUNT](#), page 78
- [COVAR_POP](#), page 79
- [COVAR_SAMP](#), page 79
- [CUME_DIST](#), page 80

- [DENSE_RANK](#), page 80
- [FIRST_VALUE](#), page 81
- [LAG](#), page 81
- [LAST_VALUE](#), page 82
- [LEAD](#), page 82
- [LISTAGG](#), page 82
- [MAX](#), page 83
- [MIN](#), page 83
- [NTH_VALUE](#), page 84
- [NTILE](#), page 84
- [PERCENT_RANK](#), page 84
- [PERCENTILE_CONT](#), page 85
- [PERCENTILE_DISC](#), page 85
- [RANK](#), page 86
- [RATIO_TO_REPORT](#), page 86
- [REGR_AVGX](#), page 86
- [REGR_AVGY](#), page 87
- [REGR_COUNT](#), page 87
- [REGR_INTERCEPT](#), page 87
- [REGR_R2](#), page 88
- [REGR_SLOPE](#), page 88
- [REGR_SXX](#), page 89
- [REGR_SXY](#), page 89
- [REGR_SYY](#), page 89
- [ROW_NUMBER](#), page 90
- [STDDEV](#), page 90
- [STDDEV_POP](#), page 91
- [STDDEV_SAMP](#), page 91
- [SUM](#), page 91
- [VAR_POP](#), page 92

- [VAR_SAMP, page 92](#)
- [VARIANCE, page 92](#)

Window Clause

More than a dozen analytical functions accept a window clause as part of ORDER BY. That capability is so noted in the sections that describe those functions. COUNT is used to illustrate how the window clause works.

The window clause has the following syntax:

```
{ {ROWS | RANGE}
  { {BETWEEN {UNBOUNDED PRECEDING | CURRENT ROW | value_expr {PRECEDING | FOLLOWING} }
    AND {UNBOUNDED FOLLOWING | CURRENT ROW | value_expr {PRECEDING | FOLLOWING} }
    |
    {UNBOUNDED PRECEDING | CURRENT ROW | value_expr PRECEDING} }
}
```

The following sections describe details of the window clause:

- [Default Assumptions, page 76](#)
- [RANGE and the Current Row, page 77](#)
- [RANGE as a Logical Offset, page 77](#)
- [ROWS and the Current Row, page 77](#)
- [ROWS and the Frame's Maximum Size, page 77](#)
- [AVG, page 78](#)

Default Assumptions

RANGE UNBOUNDED PRECEDING is assumed by default when ORDER BY is present but no window clause is supplied. For example, the following three are equivalent:

```
COUNT(*) OVER (ORDER BY hire_date)
COUNT(*) OVER (ORDER BY hire_date RANGE UNBOUNDED PRECEDING)
COUNT(*) OVER (ORDER BY hire_date RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT
ROW)
```

Similarly, the following two COUNT functions involving ROWS are equivalent:

```
COUNT(*) OVER (ORDER BY hire_date ROWS 1 PRECEDING)
COUNT(*) OVER (ORDER BY hire_date ROWS BETWEEN 1 PRECEDING AND CURRENT ROW)
```


RANGE and the Current Row

In the COUNT example below, the window frame contains the current row, all rows before it, and all ties. If the first three employees were hired on the same date, the count returned would be 3.

```
COUNT(*) OVER (ORDER BY hire_date RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
```

Likewise, when the current row moves to the second and third employees as sorted by hire date, the window frame still contains three rows, and so the result of the function is 3 in both of those cases.

As the current row advances, the resulting counts continue to track the number of employees, but if another hire-date tie occurs—for example, the ninth and tenth employees—the resulting count would be 10 for both of them.

RANGE as a Logical Offset

Because RANGE is a logical offset, the following two functions are equivalent.

The frame includes rows that are within three days of the hire date:

```
COUNT(*) OVER (ORDER BY hire_date RANGE BETWEEN 3 PRECEDING AND 3 FOLLOWING)
```

```
COUNT(*) OVER (ORDER BY hire_date RANGE BETWEEN INTERVAL '3' days PRECEDING AND INTERVAL '3' days FOLLOWING)
```

The “interval” syntax allows an expanded range of units (for example, years), and introduces more criteria for the frame size beyond row count.

ROWS and the Current Row

If ROWS is specified instead of RANGE, COUNT behaves the same as ROW_NUMBER; that is, ROWS handles only offsets of the current row. An example of such a COUNT is:

```
COUNT(*) OVER (ORDER BY hire_date ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
```

ROWS and the Frame's Maximum Size

An example of a COUNT function that limits the frame size is:

```
COUNT(*) OVER (ORDER BY hire_date ROWS BETWEEN 3 PRECEDING AND 3 FOLLOWING)
```

When the current row is the first employee, the frame size is 4 (current plus 3 following). As the current row moves through the table, the frame size can grow to 7. As the current row approaches the end of the table, the frame size goes back down to 4. With ROWS, ties have no effect on the frame size, or the resulting count.

ROWS can point outside of the data set and return results of zero. For example, the following function returns 0 when the current row is the first row of the table, because the frame is empty:

```
COUNT(*) OVER (ORDER BY hire_date ROWS BETWEEN 3 PRECEDING AND 1 PRECEDING)
```

Note: In this example, even when the current row is far enough into the table to return a nonzero count, the current row is not included, because the rows all precede the current row.

AVG

AVG returns the average of the supplied arguments.

Syntax

```
AVG ([DISTINCT | ALL ] [expression]) OVER (window_clause)
```

Remarks

- Without a window clause, AVG is a simple aggregate function. (See [AVG, page 95.](#))

CORR

CORR returns the coefficient of correlation of a set of number pairs.

Syntax

```
CORR (expression1, expression2) OVER (window_clause)
```

Remarks

- Without a window clause, CORR is a simple aggregate function.

COUNT

COUNT returns the number of rows within a partition.

Syntax

```
COUNT ( [DISTINCT | ALL ] expression) OVER (window_clause)
```

or

```
COUNT (*) OVER (window_clause)
```

Remarks

- Without a window clause, COUNT is a simple aggregate function. (See [COUNT, page 96.](#))

Example

You want to count the total number of employees by hire date. Use a query like the following:

```
COUNT (*) OVER (ORDER BY hire_date)
```

This query first orders employees by hire date, and then applies COUNT (*).

COVAR_POP

COVAR_POP returns the population covariance of a set of number pairs.

Syntax

```
COVAR_POP (expression1, expression2) [ OVER (window_clause) ]
```

Remarks

- This function takes as arguments any numeric datatype, or any nonnumeric data type that can be implicitly converted to a numeric data type.
- This function determines the argument with the highest numeric precedence, implicitly converts the remaining arguments to that datatype, and returns that datatype.
- This function follows the ANSI SQL rules for data type precedence.
- Without a window clause, COVAR_POP is a simple aggregate function.

COVAR_SAMP

COVAR_SAMP returns the covariance of a sample set of number pairs.

Syntax

```
COVAR_SAMP (expression1, expression2) OVER (window_clause)
```

Remarks

- Without a window clause, COVAR_SAMP is a simple aggregate function.

CUME_DIST

CUME_DIST calculates the cumulative distribution of a value in a group of values.

Syntax

```
CUME_DIST () OVER ( [ PARTITION BY expression [, ...] ]
ORDER BY expression [ ASC | DESC ] [ NULLS { FIRST | LAST } [, ...] ] )
```

Remarks

- CUME_DIST can be rewritten using COUNT. For example:
CUME_DIST() OVER (partition_by_order_by)

This is equivalent to either of the following COUNT expressions:

```
COUNT (*) OVER ( partition_by_order_by RANGE UNBOUNDED PRECEDING )
COUNT (*) OVER ( partition_by_order_by RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED
FOLLOWING ) )
```

- The range of values returned by CUME_DIST is (0, 1]; that is, greater than zero, and less than or equal to 1.
- Tie values always evaluate to the same cumulative distribution value.
- PARTITION BY is optional.
- ORDER BY is required.
- The window clause is not allowed.

DENSE_RANK

DENSE_RANK computes the rank of each row returned from a query with respect to the other rows, based on the values in the ORDER BY clause.

Syntax

```
DENSE_RANK () OVER ( [ PARTITION BY expression [, ...] ]
ORDER BY expression [ ASC | DESC ] [ NULLS { FIRST | LAST } [, ...] ] )
```

Remarks

- PARTITION BY is optional.
- ORDER BY is required.
- The window clause is not allowed.

FIRST_VALUE

FIRST_VALUE returns the first value in a partition.

Syntax

```
FIRST_VALUE (expression) [ (RESPECT | IGNORE) NULLS] OVER (analytic_clause)
```

Remarks

- If the first value in the set is NULL, the function returns NULL unless you specify the optional IGNORE NULLS.
- IGNORE NULLS is useful for data densification.

Example

You want to find the most senior employee for each manager in an employee table. Use a query like the following:

```
FIRST_VALUE (name) OVER (PARTITION BY manager ORDER BY hire_date)
```

This query first partitions the employees by manager, then orders employees in each partition by hire date, and then applies the FIRST_VALUE function. However, because multiple employees might have been hired on the same date, repeated execution of this query could return a different ordering of same-day hires. To make sure the returned order is consistent, add a second expression to the ORDER BY clause:

```
FIRST_VALUE (name) OVER (PARTITION BY manager ORDER BY hire_date, ID)
```

LAG

LAG provides access to more than one row of a table at the same time without a self-join. Given a series of rows returned from a query and a position of the cursor, LAG provides access to a row at a given physical offset prior to that position.

Syntax

```
LAG (expression [, offset_expression [, default_expression ] ] ) [IGNORE NULLS] OVER ([ PARTITION BY  
expression [, ...] ]  
ORDER BY expression [ ASC | DESC ] [ NULLS { FIRST | LAST } [, ...] ] )
```

Remarks

- IGNORE NULLS is optional.
- PARTITION BY is optional.

- ORDER BY is required.
- The window clause is not allowed.

LAST_VALUE

LAST_VALUE returns the last value in an ordered set of values.

Syntax

LAST_VALUE (expression) [IGNORE NULLS] OVER (window_clause)

Remarks

- If the last value in the set is NULL, the function returns NULL unless you specify IGNORE NULLS.
- IGNORE NULLS is useful for data densification.

LEAD

LEAD provides access to more than one row of a table at the same time without a self-join. Given a series of rows returned from a query and a position of the cursor, LEAD provides access to a row at a given physical offset beyond that position.

Syntax

LEAD (expression [, offset_expression [, default_expression]]) [IGNORE NULLS] OVER ([PARTITION BY
expression [, ...]]
ORDER BY expression [ASC | DESC] [NULLS { FIRST | LAST } [, ...]])

Remarks

- IGNORE NULLS and PARTITION BY are optional.
- ORDER BY is required.
- The window clause is not allowed.

LISTAGG

LISTAGG orders data within each group specified in the ORDER BY clause, and then concatenates the values of the measure column.

Syntax

LISTAGG (expression [, delimiter_expression]) WITHIN GROUP (ORDER BY expression [ASC | DESC] [NULLS { FIRST | LAST } [, ...]]) OVER (PARTITION BY expression [, ...])

Remarks

- Without an OVER clause, LISTAGG is a simple aggregate function.
- PARTITION BY is required if an OVER clause is used.

Example

```
SELECT
LISTAGG(categoryname,') WITHIN GROUP (ORDER BY categoryid) AS ALIAS
FROM
/shared/examples/ds_inventory/tutorial/categories
```

The result is:

```
alias
Data Storage,External Drives,Internal Drives,Memory,Models,Printers,Networking,Processors,Video Cards
```

MAX

MAX returns the maximum value of an expression.

Syntax

MAX ([DISTINCT | ALL] expression) OVER (window_clause)

Remarks

- Without a window clause, MAX is a simple aggregate function. (See [MAX, page 97.](#))
- The expression can be any orderable data type.

MIN

MIN returns the minimum value of an expression.

Syntax

MIN ([DISTINCT | ALL] expression) OVER (window_clause)

Remarks

- Without a window clause, MIN is a simple aggregate function. (See [MIN, page 98.](#))

- The expression can be any orderable data type.

NTH_VALUE

NTH_VALUE returns the expression value of the nth row in the window defined by the window clause. The returned value has the data type of the expression.

Syntax

NTH_VALUE (expression, nth_row) [FROM FIRST | FROM LAST] [IGNORE NULLS] OVER (window_clause)

Remarks

- FROM LAST is optional.
- If FROM LAST is not specified, FROM FIRST is the default.

NTILE

NTILE divides an ordered data set into a number of buckets indicated by expression and assigns the appropriate bucket number to each row.

Syntax

NTILE (expression1) OVER ([PARTITION BY expression [, ...]]
ORDER BY expression [ASC | DESC] [NULLS { FIRST | LAST }] [, ...])

Remarks

- The buckets are numbered 1 through expression1.
- The expression1 value must resolve to a positive constant for each partition.
- PARTITION BY is optional.
- ORDER BY is required.
- The window clause is not allowed.

PERCENT_RANK

PERCENT_RANK is similar to the CUME_DIST (cumulative distribution) function.

Syntax

PERCENT_RANK () OVER ([PARTITION BY expression [, ...]]
ORDER BY expression [ASC | DESC] [NULLS { FIRST | LAST }] [, ...])

Remarks

- The first row in any set has a PERCENT_RANK of 0.
- The range of values returned by PERCENT_RANK is 0 to 1, inclusive.
- PARTITION BY is optional.
- ORDER BY is required.
- The window clause is not allowed.

PERCENTILE_CONT

PERCENTILE_CONT is an inverse distribution function that assumes a continuous distribution model. It takes a percentile value and a sort specification, and returns an interpolated value that would fall into that percentile value with respect to the sort specification.

Syntax

```
PERCENTILE_CONT (expression) WITHIN GROUP (ORDER BY expression [ ASC | DESC ] [ NULLS { FIRST | LAST } ] [, ...] ) OVER (PARTITION BY expression [, ...] )
```

Remarks

- NULLs are ignored in the calculation.
- PARTITION BY is required if an OVER clause is used.
- Without an OVER clause, PERCENTILE_CONT is a simple aggregate function.

PERCENTILE_DISC

PERCENTILE_DISC is an inverse distribution function that assumes a discrete distribution model. It takes a percentile value and a sort specification and returns an element from the set.

Syntax

```
PERCENTILE_DISC (expression) WITHIN GROUP (ORDER BY expression [ ASC | DESC ] [ NULLS { FIRST | LAST } ] [, ...] ) OVER (PARTITION BY expression [, ...] )
```

Remarks

- Nulls are ignored in the calculation.
- PARTITION BY is required if an OVER clause is used.

- Without an OVER clause, PERCENTILE_DISC is a simple aggregate function.

RANK

RANK calculates the rank of a value in a group of values.

Syntax

```
RANK () OVER ( [ PARTITION BY expression [, ...] ]
ORDER BY expression [ ASC | DESC ] [ NULLS { FIRST | LAST } ] [, ...] )
```

Remarks

- PARTITION BY is optional.
- ORDER BY is required.
- The window clause is not allowed.

RATIO_TO_REPORT

RATIO_TO_REPORT computes the ratio of a value to the sum of a set of values. If expression1 evaluates to NULL, the ratio-to-report value also evaluates to NULL.

Syntax

```
RATIO_TO_REPORT (expression1) OVER ( [ PARTITION BY expression2 [, ...] ] )
```

Remarks

- PARTITION BY is optional.
- The window clause is not allowed.

REGR_AVGX

REGR_AVGX evaluates the average of the independent variable of the regression line.

Syntax

```
REGR_AVGX (expression1, expression2) OVER (window_clause)
```

Remarks

- The dependent variable is expression1. The independent variable is expression2.

- REGR_AVGX makes the following computation after the elimination of NULL expression1-expression2 pairs:

AVG (expression2)

- Without a window clause, REGR-AVGX is a simple aggregate function.

REGR_AVGY

REGR_AVGY evaluates the average of the dependent variable of the regression line.

Syntax

REGR_AVGY (expression1, expression2) OVER (window_clause)

Remarks

- The dependent variable is expression1. The independent variable is expression2.
- REGR_AVGY makes the following computation after the elimination of NULL expression1-expression2 pairs:

AVG (expression2)

- Without a window clause, REGR_AVGY is a simple aggregate function.

REGR_COUNT

REGR_COUNT returns an integer that is the number of non-NULL number pairs used to fit the regression line.

Syntax

REGR_COUNT (expression1, expression2) OVER (window_clause)

Remarks

- Without a window clause, REGR_COUNT is a simple aggregate function.

REGR_INTERCEPT

REGR_INTERCEPT returns the y-intercept of the regression line.

Syntax

REGR_INTERCEPT (expression1, expression2) OVER (window_clause)

Remarks

- The return value is a numeric data type and can be NULL.
- After the elimination of NULL expression1-expression2 pairs, REGR_INTERCEPT makes the following computation:

$$\text{AVG (expression1)} - \text{REGR_SLOPE (expression1, expression2)} * \text{AVG (expression2)}$$
- Without a window clause, REGR_INTERCEPT is a simple aggregate function.

REGR_R2

REGR_R2 returns the coefficient of determination (also called R-squared or goodness of fit) for the regression.

Syntax

REGR_R2 (expression1, expression2) OVER (window_clause)

Remarks

- The return value is a numeric data type and can be NULL.
- VAR_POP (expression1) and VAR_POP (expression2) are evaluated after the elimination of NULL pairs. The return values are:
 - NULL if VAR_POP (expression2) = 0
 - 1 if VAR_POP (expression1) = 0 and VAR_POP (expression2) != 0
 - POWER (CORR (expression1, expression2)) if VAR_POP (expression1) > 0 and VAR_POP (expression2) != 0
- Without a window clause, REGR_R2 is a simple aggregate function.

REGR_SLOPE

REGR_SLOPE returns the slope of a line.

Syntax

REGR_SLOPE (expression1, expression2) OVER (window_clause)

Remarks

- The return value is a numeric data type and can be NULL.
- After the elimination of NULL expression1-expression2 pairs, REGR_SLOPE makes the following computation:

$$\text{COVAR_POP}(\text{expression1}, \text{expression2}) / \text{VAR_POP}(\text{expression2})$$
- Without a window clause, REGR_SLOPE is a simple aggregate function.

REGR_SXX

REGR_SXX makes the following computation after the elimination of NULL expression1-expression2 pairs:

$$\text{REGR_COUNT}(\text{expression1}, \text{expression2}) * \text{VAR_POP}(\text{expression2})$$
Syntax

REGR_SXX (expression1, expression2) OVER (window_clause)

Remarks

- Without a window clause, REGR_SXX is a simple aggregate function.

REGR_SXY

REGR_SXY makes the following computation after the elimination of NULL expression1-expression2 pairs:

$$\text{REGR_COUNT}(\text{expression1}, \text{expression2}) * \text{COVAR_POP}(\text{expression1}, \text{expression2})$$
Syntax

REGR_SXY (expression, expression) OVER (window_clause)

Remarks

- Without a window clause, REGR_SXY is a simple aggregate function.

REGR_SYY

REGR_SYY makes the following computation after the elimination of NULL expression1-expression2 pairs:

$$\text{REGR_COUNT}(\text{expression1}, \text{expression2}) * \text{VAR_POP}(\text{expression1})$$
Syntax

REGR_SYY (expression, expression) OVER (window_clause)

Remarks

- Without a window clause, REGR_SYY is a simple aggregate function.

ROW_NUMBER

ROW_NUMBER assigns a unique number to each row to which it is applied (either each row in the partition or each row returned by the query), in the ordered sequence of rows specified in the ORDER BY clause, beginning with 1.

Syntax

```
ROW_NUMBER () OVER ( [ PARTITION BY expression [, ...] ]
ORDER BY expression [ ASC | DESC ] [ NULLS { FIRST | LAST } ] [, ...] )
```

Remarks

- PARTITION BY is optional.
- ORDER BY is required.
- The window clause is not allowed.
- If ROW_NUMBER appears in a subquery, its behavior may not be the same as the Oracle ROWNUM function.

Examples

You want to number each manager's employees by hire date. Use a query like the following:

```
SELECT
ROW_NUMBER() OVER (PARTITION BY manager ORDER BY hire_date)
FROM EMPLOYEES
```

This query first partitions the employees by manager, then orders employees in each partition by hire date, and then applies the ROW_NUMBER function. However, because multiple employees might have been hired on the same date, repeated execution of this query could return a different ordering of same-day employees. To make sure the returned order is consistent, add a second expression to the ORDER BY clause:

```
SELECT
ROW_NUMBER() OVER (PARTITION BY manager ORDER BY hire_date, ID)
FROM EMPLOYEES
```

STDDEV

STDDEV returns the sample standard deviation of expression, a set of numbers.

Syntax

STDDEV ([DISTINCT | ALL] expression) OVER (window_clause)

Remarks

- STDDEV differs from STDDEV_SAMP in that STDDEV returns zero when it has only 1 row of input data, whereas STDDEV_SAMP returns NULL.
- Without a window clause, STDDEV is a simple aggregate function.

STDDEV_POP

STDDEV_POP computes the population standard deviation and returns the square root of the population variance.

Syntax

STDDEV_POP ([DISTINCT | ALL] expression) OVER (window_clause)

Remarks

- Without a window clause, STDDEV_POP is a simple aggregate function.

STDDEV_SAMP

STDDEV_SAMP computes the cumulative sample standard deviation and returns the square root of the sample variance.

Syntax

STDDEV_SAMP ([DISTINCT | ALL] expression) OVER (window_clause)

Remarks

- Without a window clause, STDDEV_SAMP is a simple aggregate function.

SUM

SUM returns the sum of values of expression.

Syntax

SUM ([DISTINCT | ALL] expression) OVER (window_clause)

Remarks

- Without a window clause, SUM is a simple aggregate function. (See [SUM](#), page 99.)

VAR_POP

VAR_POP returns the population variance of a set of numbers after discarding the NULLs in this set.

Syntax

VAR_POP ([DISTINCT | ALL] expression) OVER (window_clause)

Remarks

- Without a window clause, VAR_POP is a simple aggregate function.

VAR_SAMP

VAR_SAMP returns the sample variance of a set of numbers after discarding the NULLs in this set.

Syntax

VAR_SAMP ([DISTINCT | ALL] expression) OVER (window_clause)

Remarks

- Without a window clause, VAR_SAMP is a simple aggregate function.

VARIANCE

VARIANCE returns the variance of expression.

Syntax

VARIANCE ([DISTINCT | ALL] expression) OVER (window_clause)

Remarks

- Without a window clause, VARIANCE is a simple aggregate function.

TDV-Supported Aggregate Functions

Aggregate functions compare or combine values in a column and return a single result based on those values.

Certain restrictions apply to the use of aggregate functions with the **DISTINCT** clause. See [DISTINCT in Aggregate Functions, page 97](#).

If any column in the **SELECT** clause is outside of an aggregate function, you must also include the column in the **GROUP BY** clause. See the example given for [AVG, page 95](#).

TDV supports the aggregate functions listed in the table below.

| TDV-Supported Aggregate Function | Comments |
|----------------------------------|---|
| AVG | See AVG, page 95 . |
| CORR | See description of the analytical form of this function, CORR, page 78 . |
| COUNT | See COUNT, page 96 . |
| COVAR_POP | See description of the analytical form of this function, COVAR_POP, page 79 . |
| COVAR_SAMP | See description of the analytical form of this function, COVAR_SAMP, page 79 . |
| LISTAGG | See description of the analytical form of this function, LISTAGG, page 82 . |
| MAX | See MAX, page 97 . |
| MEDIAN | |
| MIN | See MIN, page 98 . |
| PERCENTILE | |
| PERCENTILE_APPROX | |
| PERCENTILE_CON T | See description of the analytical form of this function, PERCENTILE_CONT, page 85 . |
| PERCENTILE_DISC | See description of the analytical form of this function, PERCENTILE_DISC, page 85 . |

| TDV-Supported Aggregate Function | Comments |
|----------------------------------|--|
| REGR_AVGX | See description of the analytical form of this function, REGR_AVGX , page 86. |
| REGR_AVGY | See description of the analytical form of this function, REGR_AVGY , page 87. |
| REGR_COUNT | See description of the analytical form of this function, REGR_COUNT , page 87. |
| REGR_INTERCEPT | See description of the analytical form of this function, REGR_INTERCEPT , page 87. |
| REGR_R2 | See description of the analytical form of this function, REGR_R2 , page 88. |
| REGR_SLOPE | See description of the analytical form of this function, REGR_SLOPE , page 88. |
| REGR_SXX | See description of the analytical form of this function, REGR_SXX , page 89. |
| REGR_SXY | See description of the analytical form of this function, REGR_SXY , page 89. |
| REGR_SYY | See description of the analytical form of this function, REGR_SYY , page 89. |
| STDDEV | See description of the analytical form of this function, STDDEV , page 90. |
| STDDEV_POP | See description of the analytical form of this function, STDDEV_POP , page 91. |
| STDDEV_SAMP | See description of the analytical form of this function, STDDEV_SAMP , page 91. |
| SUM | See SUM , page 99. |
| SUM_FLOAT | |
| VARIANCE | See description of the analytical form of this function, VARIANCE , page 92. |
| VARIANCE_POP | See description of the analytical form of this function, VAR_POP , page 92. |

| TDV-Supported Aggregate Function | Comments |
|----------------------------------|--|
| VARIANCE_SAMP | See description of the analytical form of this function, VAR_SAMP , page 92. |
| XMLAGG | See XMLAGG , page 100. |

AVG

Given a set of numeric values, AVG calculates and returns the average of the input values, as FLOAT, DECIMAL, or NULL.

Syntax

AVG (expression)

Remarks

- The expression is a numeric expression.
- AVG works only with numeric data types.
- If you want to exclude a specific row from the calculation of the average, make any column value in the row NULL.
- See [About SQL Functions in TDV](#), page 73 for an explanation of the DECIMAL(p,s) notation.

The following table lists the input types and their corresponding output types.

| Data Type of expression | Output Type |
|--|---|
| BIGINT, DOUBLE, FLOAT, INTEGER, INTERVAL_DAY, INTERVAL_YEAR, REAL, SMALLINT, TINYINT | Same type as that of the input. For example, if the input is of type TINYINT, the output is also of type TINYINT. |
| DECIMAL(p,s) NUMERIC(p,s) | DECIMAL(p,s) |
| VARCHAR | DECIMAL(p,s) Runtime exception if expression cannot be converted to a numeric value. |
| NULL | NULL |

Example

```
SELECT AVG (UnitPrice) Price, ProductID
FROM /shared/examples/ds_inventory/products products
GROUP BY ProductID
```

COUNT

COUNT counts the number of rows in a specified column or table.

Syntax

```
COUNT (expression)
COUNT (*)
```

Remarks

- The COUNT (expression) syntax specifies a column.
- The values in the specified column can be of any data type.
- The COUNT (*) syntax returns the count of all rows in a table, including NULL rows.
- If the input is a non-NULL set of values, the output is a positive integer.
- If the input is NULL, the output is zero.

The following table lists the input types that you can use in COUNT, and their corresponding output types.

| Data Type of expression | Output Type |
|---|---------------------------|
| BIGINT, BINARY, BLOB, BOOLEAN, CHAR, CLOB, DATE, DECIMAL, DOUBLE, FLOAT, INTEGER, INTERVAL_DAY, INTERVAL_YEAR, LONGVARCHAR, NUMERIC, REAL, SMALLINT, TIME, TIMESTAMP, TINYINT, VARBINARY, VARCHAR | INTEGER |
| NULL | INTEGER with a value of 0 |

Example

```
SELECT COUNT (products.ProductID) CountColumn
FROM /shared/examples/ds_inventory/products products
```

DISTINCT in Aggregate Functions

By default, aggregate functions operate on all values supplied. You can use the **DISTINCT** keyword to eliminate duplicate values in aggregate function calculations.

Note: **DISTINCT** in the **SELECT** clause and **DISTINCT** in an aggregate function do not return the same result.

To avoid misleading results from a given **SELECT** statement, do not mix aggregate functions that include a **DISTINCT** clause and aggregate functions that do not include a **DISTINCT** clause. Either all of the aggregate functions in a **SELECT** statement, or none of them, should be used with a **DISTINCT** clause.

Syntax

aggregate-function ([**ALL** | **DISTINCT**] expression)

Example

```
SELECT COUNT (DISTINCT customer_id) FROM orders
```

MAX

Given an input set of values, **MAX** returns the maximum value in that set.

Syntax

MAX (expression)

Remarks

- Expression can be numeric, string, or date-time.
- The output type is the same as the input type.
- If the input is a **CHAR**, the output is the highest string in the sorting order.
- If the input is date/time, the output is the latest date/time.
- If the input is a literal, the output is the same literal.
- If the input is a numeric expression, **MAX** compares the values in algebraic order; that is, large negative numbers are less than small negative numbers, which are less than zero.

The following table lists the input types that you can use in MAX, and their corresponding output types.

| Data Type of expression | Output Type |
|---|--|
| BIGINT, CHAR, DATE, DECIMAL, DOUBLE, FLOAT, INTEGER, INTERVAL_DAY, INTERVAL_YEAR, LONGVARCHAR, NULL, NUMERIC, REAL, SMALLINT, TIME, TIMESTAMP, TINYINT, VARCHAR | Same type as the input type. For example, if the input is of type CHAR, the output is also of type CHAR. |

Example

```
SELECT MAX (products.UnitPrice) Price,  
MAX (orders.OrderDate) Date  
FROM /shared/examples/ds_inventory/products products,  
/shared/examples/ds_orders/orders orders
```

MIN

Given an input set of values, MIN returns the minimum value in that set.

Syntax

```
MIN (expression)
```

Remarks

- The expression can be numeric, string, or date/time.
- The output type is the same as the input type.
- If the input is a CHAR, the output is the lowest string in the sorting order.
- If the input is date/time, the output is the earliest date/time.
- If the input is a literal, the output is the same literal.
- If the input is a numeric expression, MIN compares the values in algebraic order; that is, large negative numbers are less than small negative numbers, which are less than zero.

The following table lists the input types that you can use in MIN, and their corresponding output types.

| Data Type of expression | Output Type |
|---|---|
| BIGINT, CHAR, DATE, DECIMAL, DOUBLE, FLOAT, INTEGER, INTERVAL_DAY, INTERVAL_YEAR, LONGVARCHAR, NULL, NUMERIC, REAL, SMALLINT, TIME, TIMESTAMP, TINYINT, VARCHAR | Same as the input type. For example, if the input is of type TINYINT, the output is also of type TINYINT. |

Example

```
SELECT MIN (products.UnitPrice) Expr1,
MIN (orders.OrderDate) Expr2
FROM /shared/examples/ds_inventory/products products, /shared/examples/ds_orders/orders orders
```

SUM

Given a set of numeric values, SUM returns the total of all values in the input set.

Syntax

SUM (expression)

Remarks

- The expression is a numeric expression.
- SUM works only with numeric data types and data types that can be converted to numeric.
- The sum of a table with empty rows or no rows is NULL.
- See [About SQL Functions in TDV, page 73](#) for an explanation of the DECIMAL(p,s) notation.

The following table lists the input types that you can use in SUM, and their corresponding INTEGER output types.

| Data Type of expression | Output Type |
|--|---------------|
| BIGINT, DOUBLE, INTERVAL_DAY, INTERVAL_YEAR, SMALLINT, TINYINT | BIGINT |
| VARCHAR | DECIMAL(41,2) |
| FLOAT, REAL | FLOAT |

| Data Type of expression | Output Type |
|----------------------------|--|
| DECIMAL(p,s), NUMERIC(p,s) | DECIMAL (p+6, s) For example, the output of SUM(DECIMAL (4, 2)) would be SUM(DECIMAL (10, 2)) |
| NULL | NULL |

Example

```
SELECT SUM (products.UnitPrice) Total
FROM /shared/examples/ds_inventory/products products
```

XMLAGG

The XML aggregate function XMLAGG works on columns. This function is valid where other aggregate functions are valid.

This function accepts one argument, which is aggregated across the groups specified in the GROUP BY clause if that clause is specified.

Syntax

```
XMLAGG ( <XML_value_expression>
[ ORDER BY <sort_specification_list> ]
[ <XML_returning_clause> ]
)
```

Remarks

- The aggregation can be ordered with an ORDER BY clause specific to the XML aggregate function. This is independent of the SELECT ORDER BY clause.
- If the argument evaluates to NULL, the result is NULL.

Example (Without ORDER BY)

```
SELECT CAST (XMLAGG (XMLELEMENT (name Name, ContactLastName))
AS VARCHAR(10000)) "Last Name"
FROM /shared/examples/ds_orders/customers CUSTOMER
WHERE CustomerID < 23
```

Example (With ORDER BY)

```
SELECT XMLAGG ((XMLELEMENT(name Details,
XMLATTRIBUTES (ProductID as product),
XMLELEMENT (name orderno, OrderID),
XMLELEMENT (name status, Status),
XMLELEMENT (name price, UnitPrice))))
ORDER BY ProductID ASC, Status ASC, OrderID DESC, UnitPrice ASC)
```



```
myOutput
FROM /shared/examples/ds_orders/orderdetails
WHERE ProductID < 20
```

TDV-Supported Array SQL Script Functions

TDV supports the array functions listed in the table. These functions are supported in SQL scripts only and are documented in [DECLARE VECTOR](#), [page 384](#).

| TDV-Supported Array Function | Comments |
|------------------------------|----------|
| CARDINALITY | |
| CAST | |
| CONCAT | |
| EXTEND | |
| FIND_INDEX | |
| TRUNCATE | |

TDV-Supported Binary Functions

TDV supports a family of binary functions that perform bitwise logic on signed integers of length 1, 2, 4, and 8 bytes.

| Name | SQL Name | Length (bits) | Minimum | Maximum |
|------|----------|---------------|------------------------|-----------------------|
| INT1 | TINYINT | 8 | -128 | 127 |
| INT2 | SMALLINT | 16 | -32,768 | 32,767 |
| INT4 | INTEGER | 32 | -2,147,483,648 | 2,147,483,647 |
| INT8 | BIGINT | 64 | -9,223,312,036,854,776 | 9,223,312,036,854,775 |

For these functions, TDV represents values as signed integers. The leftmost bit has a value of -128; it has the dual role of designating 128 and the negative sign. All of the other bits have their customary positive value.

To determine the arithmetic value of an integer in this notation, add the values of all of the bits, with their signs:

- 1000 0000 is -128
- 1000 0001 is -127 ($1 \times -128 + 1 \times 1$)
- 1111 1110 is -2 ($1 \times -128 + 1 \times 64 + 1 \times 32 + 1 \times 16 + 1 \times 8 + 1 \times 4 + 1 \times 2$)

TDV supports the binary functions listed in the table.

| TDV-Supported Binary Function | Comments |
|------------------------------------|---|
| INT1AND, INT2AND, INT4AND, INT8AND | See AND Functions, page 102 |
| INT1NOT, INT2NOT, INT4NOT, INT8NOT | See NOT Functions, page 103 |
| INT1OR, INT2OR, INT4OR, INT8OR | See OR Functions, page 104 |
| INT1SHL, INT2SHL, INT4SHL, INT8SHL | See SHL Functions, page 105 |
| INT1SHR, INT2SHR, INT4SHR, INT8SHR | See SHR Functions, page 106 |
| INT1XOR, INT2XOR, INT4XOR, INT8XOR | See XOR Functions, page 108 |

AND Functions

The AND functions create a result by combining each bit of one number with the corresponding bit of the other number. If a pair of corresponding bits are both 1, the result for that bit position is 1; otherwise the result is 0, as shown in the table.

| AND | arg1 | |
|------|------|---|
| | 0 | 1 |
| arg2 | 0 | 0 |
| | 1 | 0 |

Sample Syntax
INT1AND(arg1, arg2)

Remarks

- The AND functions are commutative; that is, the order of the arguments does not affect the outcome.

Examples

| Function Input | Result | Comments |
|------------------|--------|---|
| INT1AND(0,x) | 0 | 0 ANDed with any integer returns 0. |
| INT1AND(-0,x) | 0 | -0 is mapped to 0 before ANDing it with the other argument. |
| INT1AND(-64,64) | 64 | |
| INT1AND(-64,66) | 64 | |
| INT1AND(-1,127) | 127 | -1 is represented by all 1-bits, so it returns any number it is ANDed with. |
| INT1AND(-128,-x) | -128 | -128 ANDed with any negative integer (except -0) returns -128. |

NOT Functions

The NOT functions change each 1 to a 0 and each 0 to a 1 in the binary representation of the argument.

Sample Syntax

INT1NOT(arg)

Remarks

- As long as the argument value is in range of the function, the returned value is the same for INT1NOT, INT2NOT, INT4NOT, and INT8NOT. For example, INT1NOT(-127) = INT2NOT(-127) = INT4NOT(-127) = INT8NOT(-127).
- Both 0 and -0 inputs return -1, but -1 input returns only 0.

Examples

The table shows representative input and output values for the INT1NOT function.

| Function Input | Result |
|----------------|--------|
| INT1NOT(0) | -1 |
| INT1NOT(1) | -2 |
| INT1NOT(2) | -3 |
| ... | |
| INT1NOT(126) | -127 |
| INT1NOT(127) | -128 |
| INT1NOT(-128) | 127 |
| INT1NOT(-127) | 126 |
| ... | |
| INT1NOT(-2) | 1 |
| INT1NOT(-1) | 0 |
| INT1NOT(-0) | -1 |

OR Functions

The OR functions create a result by combining each bit of one number with the corresponding bit of the other number. If a pair of corresponding bits are both 0, the result for that bit position is 0; otherwise the result is 1, as shown in the table.

| OR | arg1 | | |
|------|------|---|---|
| | 0 | 1 | |
| arg2 | 0 | 0 | 1 |
| | 1 | 1 | 1 |

Sample Syntax

INT1OR(arg1, arg2)

Remarks

- The OR functions are commutative; that is, the order of the arguments does not affect the outcome.

Examples

| Function Input | Result | Comments |
|------------------|--------|---|
| INT1OR(0,x) | x | 0 ORed with any number returns the same number, regardless of sign. |
| INT1OR(-0,x) | x | -0 is mapped to 0 before being ORed with the other argument. |
| INT1OR(64,-64) | -64 | |
| INT1OR(64,-66) | -2 | |
| INT1OR(66,-64) | -62 | |
| INT1OR(-66,-64) | -2 | |
| INT1OR(-1,x) | -1 | -1 ORed with any positive number results in -1. |
| INT1OR(-128,1) | -127 | |
| ... | | |
| INT1OR(-128,127) | -1 | |
| INT1OR(-128,-x) | -x | -128 ORed with any negative number results in the same negative number. |

SHL Functions

The SHL functions left-shift the bits of the binary representation of a number.

Sample Syntax

```
INT1SHL(arg1, arg2[, arg3])
```

Remarks

- Shifts arg1 left by arg2 bits, filling with zeros on the right.
- If arg3 is present, arg1 is ANDed with arg3 before being shifted.

- Each left bit-shift doubles the number.

Examples

The table below shows examples of SHL. Most of the examples use INT1.

| Function Input | Result | Comments |
|------------------|--------|--|
| INT1SHL(1,0) | 1 | Arg2 is 0, so no shift takes place. |
| INT1SHL(1,1) | 2 | |
| INT1SHL(3,2) | 12 | |
| INT1SHL(3,10) | 12 | Arg2 is 10, the same as 2 mod 8 (the number of bits in INT1), so the result is the same as INT1SHL(3,2). |
| INT1SHL(27,1,14) | 20 | Arg3 is present. 27 (0001 1011) is ANDed with 14 (0000 1110), with result 10 (0000 1010). Shifted left 1, it becomes 20 (0001 0100). |
| INT1SHL(127,1) | | |
| INT2SHL(127,17) | | |
| INT1SHL(-2,1) | | |
| INT1SHL(-127,0) | | |
| INT1SHL(-127,1) | 2 | |
| INT1SHL(-128,0) | 0 | |
| INT2SHL(-128,0) | | |

SHR Functions

The SHR functions right-shift the bits of the binary representation of a number.

Sample Syntax

INT1SHR(arg1, arg2[, arg3])

Remarks

- Shifts arg1 right by arg2 bits.

- With each shift, a 0 is placed in the second-most-significant bit of the INTEGER (of whatever size), and the least significant bit is shifted out.
- If arg3 is present, arg1 is ANDed with arg3 before being shifted.
- Each left bit-shift doubles the number.
- The most significant bit of the binary representation of arg1 acts like a sign bit. It does not move or change; that is, negative numbers remain negative, and positive numbers remain positive.
- If arg1 is an odd number (whether positive or negative), the result of each position shift is (arg1 minus 1) divided by 2. If arg1 is even, the result is arg1 divided by 2.
- Arg2 should be a nonnegative number (positive or 0).

Examples

The table below shows examples of SHR. Most of the examples use INT1.

| Function Input | Result | Comments |
|------------------|--------|---|
| INT1SHR(1,0) | 1 | Arg2 is 0, so no shift takes place. |
| INT1SHR(1,1) | 0 | |
| INT1SHR(2,1) | 1 | |
| INT1SHR(3,1) | 1 | Adjacent pairs of arg1 values map to the same result. |
| INT1SHR(5,1) | 2 | 5 is odd, so the result is $5 - 1 (=4)$ divided by 2, or 2. |
| INT1SHR(-5,1) | -3 | -5 is odd, so the result is $-5 - 1 (= -6)$ divided by 2, or -3. |
| INT2SHR(127,1) | 63 | |
| INT2SHR(127,1,6) | 3 | Arg3 is present. Because both the 4-bit and the 2-bit are set in 127 (0111 1111), the AND result is 6; shifted right one position it becomes 3. |
| INT2SHR(127,17) | 63 | Arg2 is 9, the same as 1 mod 16 (the number of bits in INT2), so the result is the same as INT2SHR(127,1). |
| INT1SHR(-128,8) | -128 | Arg 2 is 8, the same as 0 mod 8, so the result is the same as INT1SHR(-128,0); that is, no shift. |

XOR Functions

The XOR (exclusive-OR) functions create a result by combining each bit of one number with the corresponding bit of the other number. If a pair of corresponding bits are the same, the result for that bit position is 0; if they are different, the result is 1, as shown in the table.

| XOR | arg1 | | |
|------|------|---|---|
| | 0 | 1 | |
| arg2 | 0 | 0 | 1 |
| | 1 | 1 | 0 |

Sample Syntax

INT1XOR(arg1, arg2)

Remarks

- The XOR functions are commutative; that is, the order of the arguments does not affect the outcome.

Examples

| Function Input | Result | Comments |
|------------------|--------|--|
| INT1XOR(0,x) | x | 0 has no bits set, so every bit set in x is set in the result. |
| INT1XOR(0,x) | -x | -0 is mapped to 0 before being XORed to arg2. |
| INT1XOR(-0,-x) | x | -0 is mapped to 0 before being XORed to arg2. |
| INT1XOR(64,-64) | -128 | |
| INT1XOR(64,-66) | -2 | |
| INT1XOR(66,-64) | -126 | |
| INT1XOR(-66,-64) | 126 | |
| INT1XOR(-1,127) | -128 | |
| INT1XOR(-128,1) | -127 | |

| Function Input | Result | Comments |
|--------------------|--------|----------|
| ... | | |
| INT1XOR(-128,127) | -1 | |
| INT1XOR(-128,-127) | 1 | |
| ... | | |
| INT1XOR(-128,-1) | 127 | |

TDV-Supported Character Functions

Character functions let you get information about strings, combine them, or modify them.

TDV supports the character functions listed in the table.

| TDV-Supported Character Function | Comments |
|----------------------------------|--|
| ASCII | See ASCII , page 112 |
| BITCOUNT | |
| BITSTREAM_TO_BINARY | |
| BIT_LENGTH | |
| BTRIM | |
| CHAR_LENGTH | |
| CHR | See CHR , page 112 |
| CONCAT | See CONCAT , page 113 |
| DLE_DST | |
| FIND | See INSTR , page 116. To use the syntax for FIND, replace <INSTR> with FIND. |
| FIND_IN_SET | |
| GET_JSON_OBJECT | See GET_JSON_OBJECT , page 115 |

| TDV-Supported Character Function | Comments |
|----------------------------------|--|
| GREATEST | GREATEST treats empty strings as NULL. |
| HEX_TO_BINARY | |
| INET_ATON | |
| INET_NTOA | |
| INITCAP | |
| INSTR | See INSTR , page 116 |
| ISUTF8 | |
| JSON_TABLE | See JSON_TABLE , page 167 |
| LCASE | |
| LEAST | LEAST treats empty strings as NULL. |
| LEFT | |
| LENGTH | See LENGTH , page 117 |
| LE_DST | |
| LOWER | See LOWER , page 118 |
| LPAD | See LPAD , page 119 |
| LTRIM | |
| MD5 | |
| OCTET_LENGTH | |
| OVERLAYB | |
| PARSE_URL | |
| PARTIAL_STRING_MASK | See PARTIAL_STRING_MASK , page 120 |
| POSITION | See POSITION , page 121 |
| QUOTE_IDENT | |

| TDV-Supported Character Function | Comments |
|----------------------------------|---|
| QUOTE_LITERAL | |
| REGEXP_EXTRACT | |
| REGEXP_REPLACE | |
| REPEAT | |
| REPLACE | See REPLACE , page 122 |
| REVERSE | |
| RIGHT | |
| RPAD | See RPAD , page 123 |
| RTRIM | See RTRIM , page 124 |
| SPACE | See SPACE , page 125 |
| SOUNDEX | |
| SPLIT_PART | |
| STRPOS | |
| SUBSTR | See SUBSTR and SUBSTRING , page 126 |
| TO_CANONICAL | |
| TRANSLATE | |
| TRIM | See TRIM , page 128 |
| TRIMBOTH | |
| TRIMLEADING | |
| TRIMTRAILING | |
| UCASE | |
| UNICHR | |
| UNICODE | |

| TDV-Supported Character Function | Comments |
|----------------------------------|--------------------------------------|
| UPPER | See UPPER , page 129 |
| V6_ATON | |
| V6_NTOA | |
| V6_SUBNETA | |
| V6_SUBNETN | |
| V6_TYPE | |

ASCII

ASCII returns the numerical value of an ASCII character.

Syntax

ASCII (expression)

Remarks

- If you pass a NULL string to this function, it returns 0.
- If the string is empty, this function returns 0.
- Any character outside the range 0 to 255 is returned as an error or ignored, depending on the implementation of RDBMS.
- If expression is a string with more than one character, only the first character is considered.

Example

```
SELECT ASCII('a') AS lowercase_a,  
       ASCII('A') AS uppercase_a
```

CHR

CHR converts an integer ASCII code to a character.

Syntax

CHR (integer)

Remarks

- CHR can accept string input, as long as the string can be converted to a numeric value.
- The input must be a value between 0 and 255, inclusive.
- If the input is NULL, the output is NULL.
- If the input is less than zero, an exception is thrown.
- If the input is greater than the maximum value of INTEGER (2147483647), an exception is thrown.
- For an ASCII chart, see <http://www.techonthenet.com/ascii/chart.php>

The following table lists the input types that you can use in CHR, and their corresponding output types.

| Data Type of integer | Output Type |
|---|-------------|
| BIGINT, DECIMAL, INTEGER, SMALLINT, STRING, TINYINT | CHAR(1) |
| NULL | NULL |

Example

```
SELECT DISTINCT CHR (100)
FROM /shared/examples/ds_orders/customers
```

CONCAT

Given two arguments, the CONCAT function concatenates them into a single output string.

Note: You can also concatenate two arguments in-line using the concatenation operator (||); for example, A || B.

Syntax

```
CONCAT (argument1, argument2)
```

Remarks

- The arguments of CONCAT can be of type string or any other type, and you can concatenate them in any combination of data types.

- To concatenate a nonstring to a string, use the CAST function to convert the nonstring to string.
- Enclose a literal string within single-quotes to concatenate it with another argument. For example, CONCAT('string1', string2), where string1 is a literal.
- The CONCAT function does not supply white-space characters between arguments in the concatenated output. You must provide the white-space characters manually.

You can use the Subfunction button in the Function Arguments Input dialog to provide a space between concatenated strings, or use the format:

```
CONCAT('string1', CONCAT(' ', 'string2'))
```

- If any of the input strings in a CONCAT function is NULL, the result string is also NULL. Otherwise, the output type is STRING.

The following table lists the input types that you can use in CONCAT.

| Data Type of argument1 | Data Type of argument2 | Output Type |
|--|---|-------------|
| BIGINT, CHAR, DATE, DECIMAL, FLOAT, INTEGER, LONGVARCHAR, NUMERIC, REAL, SMALLINT, STRING, TIME, TIMESTAMP, TINYINT, VARCHAR | Any type listed for argument1 except NULL.n | STRING |
| Any data type listed above. | NULL | NULL |
| NULL | | NULL |

Examples (Generic)

```
CONCAT (<string>, <string>)
CONCAT (<string>, <nonstring>)
CONCAT (<nonstring>, <string>)
CONCAT (<nonstring>, <nonstring>)
```

Examples (Specific)

```
SELECT CONCAT (customers.ContactFirstName,
    CONCAT (' ', customers.ContactLastName)) Expr1,
    CONCAT ('a', concat(' ', 'b')) Expr2,
    CONCAT ('a', concat(' ', NULL)) Expr3,
    CONCAT ('NULL', concat(' ', NULL)) Expr4,
    CONCAT (NULL, concat(' ', NULL)) Expr5,
    CONCAT ('a', current_date) Expr6,
    CONCAT (current_date, current_time) Expr7,
    CONCAT ('Feb', concat(' ', CAST(2004 AS BIT))) Expr8,
    customers.ContactFirstName || ' ' ||
    customers.ContactLastName Expr9,
```

```
'0100' || '1010' Expr10, 100 || 1010 Expr11, 23 || 56 Expr12
FROM /shared/examples/ds_orders/customers customers
```

GET_JSON_OBJECT

GET_JSON_OBJECT is a push-only function that extracts a JSON object from a JSON string based on the JSON path, and returns a JSON string of the extracted JSON object.

Syntax

```
GET_JSON_OBJECT (STRING json_string, STRING json_path)
```

Remarks

- The json_path argument can contain only numbers, lowercase letters, and underscore (_).
- Keys cannot start with numbers because of restrictions on Hive/Hadoop column names.
- This function does not support recursive descent using '..'.
- This function does not support filter expression '[?(<expression>)]'.
- Return value is NULL if the input JSON string is invalid.
- Union operator and array slice operator is not supported by this function.

Examples

The following is a simple example that uses GET_JSON_OBJECT.

```
PROCEDURE JSONPathFunctionExample(OUT resultJson VARCHAR)
BEGIN
  DECLARE sourceJson VARCHAR(4096);
  DECLARE jsonPathExpression VARCHAR(4096);

  --Create a JSON value to use in the JSONPATH function
  SET sourceJson = '{"LookupProductResponse":{"LookupProductResult":{"row":[
{"ProductName":"Maxtific 40GB ATA133 7200","ProductID":"1","ProductDescription":"Maxtific Storage 40 GB"}
]}}';

  --Create a JSONPATH expression to evaluate
  SET jsonPathExpression = '$.LookupProductResponse.LookupProductResult.row[0].ProductName';

  --Evaluate the XPATH expression against the source XML value
  SET resultJson = JSONPATH (sourceJson, jsonPathExpression);
END
```

The output of this example is 'Maxtific 40GB ATA133 7200'.

You can also use `GET_JSON_OBJECT` to iterate through an array and count the elements.

```
SET i = 0;
SET jsonobject = GET_JSON_OBJECT(jsonstring,'$.array_element[\"||CAST(i AS VARCHAR)||\"]');
WHILE jsonobject NOT NULL DO
SET i = i + 1 ;
SET jsonobject = GET_JSON_OBJECT(jsonstring,'$.array_element[\"||CAST(i AS VARCHAR)||\"]');
END DO;
```

INSTR

The `INSTR` (“in string”) function searches for a character or substring within a string and returns an integer for the location if that string is found, or zero if it is not found. The first argument, which can be a literal string, a variable, or a table column, is searched for the string specified by the second argument. If the string is found within the string, its position is returned as an integer relative to either the start or the end of the string.

Syntax

```
INSTR (string_to_examine, string_to_find[, search_start[, nth_occurrence]])
```

Remarks

- The first argument, `string_to_examine`, can be a literal expression or variable name enclosed in single-quotes. The first argument can also be an expression within a SQL `SELECT` to evaluate the values within a `tableName.columnName`. The data type must be `VARCHAR` or similar.
- The second argument, `string_to_find`, should be a string, or a variable with a data type of `VARCHAR`.
- Optionally, you can specify `search_start` to make the search proceed from any arbitrary position within the string.
- If the search proceeds from the end of `string_to_examine`, the result is always 0.
- If `INSTR` is executed in TDV, it returns `NULL` for `INSTR('','C')` and 0 for `INSTR(' ','C')`. When pushed to some databases, `INSTR('','C')` might return 0 as opposed to `NULL`.

Note: The difference is a space character. The C character is just an example.

- `INSTR` treats empty strings as `NULL`.
- The location of any substring match is reported with a count that starts with the first character position on the left.

- The INSTR function can be used to parse a concatenated value to identify the spaces between space-delimited names or words.
- Each leading space counts as one character.

Note: See also the related function [POSITION](#), page 121.

Examples

```
INSTR(' jean_doe', ' ', 2, 1)
```

This sample INSTR function call (with one leading space) returns 6.

```
INSTR(' jean_doe', ' ', 2, 1)
```

This sample INSTR function call (with two leading spaces) returns 2.

LENGTH

LENGTH returns the number of characters (rather than the number of bytes) in a given string expression.

Syntax

```
LENGTH (string)
```

Remarks

- CHAR_LENGTH and CHARACTER_LENGTH are synonymous with LENGTH.
- If the input is NULL, the output is also NULL. Otherwise, the output is an integer that is equal to or greater than zero.
- If the input is an empty string, the output is zero.
- The length of a white-space in an input argument is counted as 1 (one).
- If you want to count the white-space included in an input string, use the CONCAT function to accommodate the space, as in this example:

```
LENGTH (CONCAT (customers.ContactFirstName, CONCAT (' ', customers.ContactLastName)))
```

- If you want to find the length of an integer, you must convert the integer to VARCHAR and then pass the string as the input for the LENGTH function.

For example, if you want to find out the number of digits in a phone number, cast the phone number's integer into a VARCHAR and use it in the LENGTH function.

The following table lists the input types that you can use in LENGTH, and their corresponding output types.

| Data Type of string | Output Type |
|--|-------------|
| BLOB, CHAR, CLOB, LONGVARCHAR, VARCHAR | INTEGER |
| NULL | NULL |

Example

```
SELECT LENGTH (customers.PostalCode) Expr1,  
LENGTH (NULL) Expr2,  
LENGTH ( ' ') Expr3,  
LENGTH (") Expr4,  
LENGTH (CONCAT(customers.ContactFirstName,  
CONCAT(' ', customers.ContactLastName))) Expr5,  
LENGTH (customers.FaxNumber) Expr6,  
LENGTH (TO_CHAR(1000)) Expr7,  
LENGTH (CAST (customers.PhoneNumber AS VARCHAR)) Expr8  
FROM /shared/examples/ds_orders/customers customers
```

LOWER

The LOWER function makes all the alphabetical characters in a given string lowercase. It can be used to format output, or to make case-insensitive comparisons.

Syntax

```
LOWER (string)
```

Remarks

- The input string must be enclosed within single-quotes.
- If the input is an empty string, the output is also an empty string.
- If the input contains only space characters enclosed in single-quotes, it is not empty, and LOWER does not turn it into an empty string.

The following table lists the input types that you can use in LOWER, and their corresponding output types.

| Data Type of string | Output Type |
|------------------------------------|---|
| CHAR, LONGVARCHAR, STRING, VARCHAR | Same as the input type; for example, if the input is of type VARCHAR, the output is also of type VARCHAR. |

| Data Type of string | Output Type |
|---------------------|-------------|
| NULL | NULL |

Example (With a Comparison)

```
SELECT ContactLastName AS Name
FROM /shared/examples/ds_orders/customers
WHERE LOWER (ContactLastName) LIKE '%Ho%';
```

This example would convert all the letters in a ContactLastName to lowercase and pull out all the names from the table customers containing the sequence ho, such as:

```
Howard
Honner
Nicholson
Thompson
```

Example (Other Contexts)

```
SELECT LOWER (products.ProductName) Name,
LOWER ('YOU') Expr4,
LOWER (' ') Expr6,
LOWER ('YoU 9 fEEt') Expr2,
LOWER (NULL) Expr1
FROM /shared/examples/ds_inventory/products products
```

LPAD

The LPAD function truncates strings from the right, or pads them with spaces (or specified characters) on the left, to make all returned values the same specified length.

Syntax

```
LPAD (expression, padded_length [, pad_string])
```

Remarks

- The expression argument can be a literal, a variable set off by single-quotes, or a SQL expression specifying table.columnName. The data type of the column specified must be compatible with VARCHAR or a related data type, but not INTEGER, TINYINT, or CHAR(1).
- If expression is an empty string or a NULL string, LPAD returns NULL.
- The padded_length argument is an integer that specifies the length of the returned values.

- If `padded_length` is zero or negative, `LPAD` returns an empty string.
- The `pad_string` argument is optional. If it is omitted, spaces are used as the left-padding character; otherwise, `pad_string` is added repeatedly as left-padding until the return value reaches the specified integer string length, as shown in the fourth example below.
- If `pad_string` is an empty string or a `NULL` string, `LPAD` returns `NULL`.

Note: See also the related function [RPAD](#), page 123.

Example (Retrieve the First Character)

The following SQL example uses `LPAD` to retrieve just the first character from the values in the column `FirstName`.

```
SELECT LPAD (table.FirstName, 1) FirstInitial FROM table
```

Example (Truncate Values)

The following SQL example uses `LPAD` to truncate the values from the `FamilyName` column so that only the first twelve characters from very long family names are returned in the result set column that has the alias `LastName(12)`.

```
SELECT LPAD (table.FamilyName, 12) LastName(12) FROM table
```

Example (Limit Values or Left-Pad with a Value)

The following SQL example uses `LPAD` to limit the values of `SectionTitle` to the first 36 characters, and to precede section titles of fewer than 36 characters with enough periods to bring their character counts to 36.

```
SELECT LPAD (table.SectionTitle, 36, '.') FROM table
```

Example (Limit Values or Left-Pad with a Pattern of Values)

When `pad_string` is more than a single character, the specified character pattern (or beginning of the pattern) is repeated as padding until the exact string length is reached.

```
SELECT LPAD (table.LastName, 8, '*...') FROM table
```

In this example, a last name of “Shimabukuro” would return “Shimabuk” and a last name of “Ho” would return “*...*.Ho”.

PARTIAL_STRING_MASK

This string masking function provides the ability to reveal the first and the last few specified number of characters with a custom padding string in the middle.

Syntax

`partial_string_mask(<str>, <prefix>, <padding>, <suffix>)`

Remarks

- <str> is the string to be masked.
- <prefix> is the starting number of characters to be revealed.
- <padding> is the custom padding string in the middle.
- <suffix> is the last number of characters to be revealed from the column value.

POSITION

Given two input expressions, the POSITION function returns an integer value representing the starting position of the first expression within the second expression.

Syntax

`POSITION (expression1 IN expression2)`

Remarks

- This function uses the case-sensitivity setting of the TDV Server (TDV Server > SQL Engine > SQL Language > Case Sensitivity).
- POSITION accepts all string types and all numeric types as input arguments.
- The output is always an integer, provided that none of the input strings is NULL. Otherwise, NULL is returned.
- If either argument is NULL, the function returns NULL.
- If the first argument is a blank string, the function returns 1 (one).
- If the first argument is not found within the second argument, the function returns zero.

Note: See also the related function [INSTR](#), page 116.

Examples

`POSITION ('ec' IN 'lecture')`

The output returned is 2, because ec starts at the second character position of expression2.

`POSITION (' ' IN 'lecture')`

The output returned is 0 because expression2 does not contain a space character.
`POSITION (" IN 'lecture')`

The output returned is 1 because expression1 is the empty string.

REPLACE

Given a series of three strings (representing the search string, string to be replaced, and replacement string, respectively), the REPLACE function substitutes the replacement string for all instances of the string to be replaced that are contained in the search string.

Syntax

`REPLACE (search_string, string_to_be_replaced, replacement_string)`

Remarks

- The string_to_be_replaced and the replacement_string must be of the same type (string or binary).
- All occurrences of the string_to_be_replaced within the search_string are replaced with the replacement_string.
- The string_to_be_replaced and the replacement_string must be enclosed within single-quotes.
- If any of the input strings is NULL, the output is also NULL. Otherwise, the output is a string.

The following table lists the input types that you can use in REPLACE, and their corresponding output types.

| Data Type of search_string | Data Type of string_to_be_replaced | Data Type of replacement_string | Output Data Type |
|--|------------------------------------|---------------------------------|--------------------------------|
| CHAR, VARCHAR, LONGVARCHAR, STRING | Same as search_string. | Same as string_to_be_replaced. | Same as string_to_be_replaced. |
| CHAR, LONGVARCHAR, NULL, STRING, VARCHAR | NULL | Same as search_string. | NULL |

| Data Type of search_string | Data Type of string_to_be_replaced | Data Type of replacement_string | Output Data Type |
|------------------------------------|------------------------------------|---------------------------------|------------------|
| NULL | CHAR, VARCHAR, LONGVARCHAR, STRING | Same as string_to_be_replaced. | NULL |
| CHAR, LONGVARCHAR, STRING, VARCHAR | Same as search_string. | NULL | NULL |

Example

```
SELECT REPLACE (products.ProductName, 'USB 2.0', 'USB 3.0') Replaced
FROM /shared/examples/ds_inventory/products products
```

RPAD

The RPAD function truncates strings from the right, or pads them with spaces (or specified characters) on the right, to make all returned values the same specified length.

Syntax

RPAD (expression, padded_length [, pad_string])

Remarks

- The expression argument can be a literal expression, a variable set off by single-quotes, or a SQL expression specifying table.columnName. The data type of the column specified must be compatible with VARCHAR or a related data type, but not INTEGER, TINYINT, or CHAR(1).
- If expression is an empty string or a NULL string, RPAD returns NULL.
- The padded_length argument is an integer that specifies the length of the returned values.
- If padded_length is zero or negative, RPAD returns an empty string.
- The pad_string argument is optional. If it is omitted, spaces are used as the right-padding character; otherwise, pad_string is added repeatedly on the right until the return value reaches the specified string length, as shown in the fourth example below.
- If pad_string is an empty string or a NULL string, RPAD returns NULL.

Note: See also the related function [LPAD](#), page 119.

Example (Retrieve the First Character)

The following SQL select uses RPAD to retrieve just the first two characters from the values in the column FirstName.

```
SELECT RPAD (table.FirstName, 2) FirstInitial FROM table
```

Example (Truncate Values)

The following SQL select uses RPAD to truncate the values from the FamilyName column so that only the first twelve characters from very long family names are returned in the result column that has the alias LastName(12).

```
SELECT RPAD (table.FamilyName, 12) LastName(12) FROM table
```

Example (Limit Values or Right-Pad with a Value)

The following SQL select uses RPAD to limit the values of SectionTitle to the first 36 characters, and to append enough periods to shorter section titles to bring their character counts to 36.

```
SELECT RPAD (table.SectionTitle, 36, '.') FROM table
```

Example (Limit Values or Right-Pad with a Pattern of Values)

When pad_string is more than a single character, the specified characters are repeated as padding until the length specified by padded_length is reached.

```
SELECT RPAD (table.LastName, 10, '*...') FROM table
```

In this example, a LastName of “Shimabukuro” would return “Shimabuk”; a LastName of “Ho” would return “Ho*...*...” (that is, with all or part of the pattern asterisk-dot-dot-dot repeated until a count of 10 characters has been reached).

RTRIM

The RTRIM function trims all white-spaces from the right side of a string.

Syntax

```
RTRIM (string) [ ]
```

Remarks

- White-spaces embedded in an input string are not affected.
- If the input string is NULL, the output is also NULL. Otherwise, the output is of the same type as the input.

The following table lists the input types that you can use in RTRIM, and their corresponding output types.

| Data Type of string | Output Type |
|----------------------------------|--|
| CHAR, LONGVARCHAR, NULL, VARCHAR | Same type as the input type. For example, if the input is of type CHAR, the output is also of type CHAR. |

Example (No White-Space before Second Concatenated String)

```
concat (RTRIM ('AAA '), 'Member')
```

This example has white-spaces at the end of the sequence AAA and no white-space character preceding the M in Member. It produces the following result:

```
AAAMember
```

Example (White-Space before Second Concatenated String)

```
concat (RTRIM ('AAA '), ' Member')
```

This example has white-spaces at the end of the sequence AAA and one white-space character preceding the M in Member. It produces the following result:

```
AAA Member
```

SPACE

The SPACE function returns a string of as many spaces as the integer specifies.

Syntax

```
SPACE (integer)
```

Remarks

- This function accepts a DECIMAL input value.
- If the input is NULL, the output is also NULL; otherwise, the output is a string.
- If the input is a negative integer, the output is NULL.

The following table lists the input types that you can use in SPACE, and their corresponding output types.

| Data Type of integer | Output Type |
|---|-------------|
| BIGINT, DECIMAL, INTEGER, SMALLINT, TINYINT | CHAR |
| NULL | NULL |

Example

```
SELECT CONCAT (customers.ContactFirstName,  
CONCAT (SPACE (1), customers.ContactLastName)) Name  
FROM /shared/examples/ds_orders/customers customers
```

SUBSTR and SUBSTRING

Given a string, the SUBSTR and SUBSTRING functions return the substring starting from the start position, and extending up to the length specified by the substring length.

Syntax

```
SUBSTR (string, start_position, length_of_substring)  
SUBSTRING (string, start_position, length_of_substring)
```

- Remarks**
- Start_position and length_of_substring must be positive integers.

Note: Results might differ between pushed and not pushed if the second argument is negative.
 - The original string is assumed to start at position one (1).
 - The resulting substring is any sequence of characters in the original string, including an empty string.
 - If the original string is an empty string, the resulting substring is also an empty string.
 - If any of the input arguments is NULL, the output is also NULL.

The following table lists the input types that you can use in SUBSTRING, and their corresponding output types.

| Data Type of string | Data Type of start_position | Data Type of length_of_substring | Data Type of Output |
|---------------------|-----------------------------|----------------------------------|--------------------------|
| CHAR | TINYINT | Same as start_position. | Same as string argument. |
| LONGVARCHAR | INTEGER | | |
| STRING | BIGINT | | |
| VARCHAR | SMALLINT | | |
| NULL | BIGINT | Same as start_position. | NULL |
| | INTEGER | | |
| | NULL | | |
| | SMALLINT | | |
| | TINYINT | | |
| CHAR | NULL | TINYINT | NULL |
| LONGVARCHAR | | INTEGER | |
| STRING | | BIGINT | |
| VARCHAR | | SMALLINT | |
| CHAR | TINYINT | NULL | NULL |
| LONGVARCHAR | INTEGER | | |
| STRING | BIGINT | | |
| VARCHAR | SMALLINT | | |

Example

```
SELECT SUBSTRING (customers.PhoneNumber, 1, 5) AreaCode
```

TRIM

The TRIM function removes all instances of some specified character (default: blanks) from the input string. By default, TRIM removes the character from the beginning and end of the input string (BOTH). TRIM can remove the character from just the beginning of the string (LEADING) or the end of the string (TRAILING).

Syntax

TRIM ([[BOTH | LEADING | TRAILING] [character_to_trim] FROM] string)

Remarks

- If the input string is NULL, the output is also NULL. Otherwise, the output is a string.
- If you also want to trim characters within a string, use the REPLACE function. (See [REPLACE](#), page 122.)
- When no character to trim is specified, the TRIM function removes ASCII space characters (value 32), but not Unicode nonbreaking space characters (value 160).

The following table lists the valid input types, and their corresponding output types.

| Data Type of string | Output Type |
|----------------------------------|------------------------------|
| CHAR, LONGVARCHAR, VARCHAR, NULL | Same as the input data type. |

Examples

This example removes all leading and trailing ASCII space characters from the string, resulting in 'ababa':

```
SELECT TRIM (' ababa ')
FROM /services/databases/system/DUAL
```

This example is equivalent to the one above:

```
SELECT TRIM (BOTH ' ababa ')
FROM /services/databases/system/DUAL
```

This TRIM function results in bab:

```
SELECT TRIM (BOTH 'a' FROM 'ababa')
FROM /services/databases/system/DUAL
```

This TRIM function results in baba:

```
SELECT TRIM (LEADING 'a' FROM 'ababa')
FROM /services/databases/system/DUAL
```

This TRIM function results in abab:

```
SELECT TRIM (TRAILING 'a' FROM 'ababa')
FROM /services/databases/system/DUAL
```

UPPER

The UPPER function returns the specified string with all alphabetical characters uppercase. It can be used to format output, or to make case-insensitive comparisons.

Syntax

UPPER (string)

Remarks

- The input string must be enclosed within single-quotes.
- If the input is an empty string, the output is also an empty string.
- If the input contains only space characters enclosed in single-quotes, it is not empty, and UPPER does not turn it into an empty string.

The following table lists the input types that you can use in UPPER, and their corresponding output types.

| Data Type of string | Output Type |
|-------------------------------------|--------------------|
| CHAR, LONGVARCHAR, NULL, VARCHAR | Same as the input. |

Example

```
SELECT UPPER (products.ProductName) ProductName
FROM /shared/examples/ds_inventory/products products
```

TDV-Supported Conditional Functions

TDV supports the conditional functions listed in the table.

| TDV-Supported Conditional Function | Comments |
|------------------------------------|--|
| COALESCE | See COALESCE , page 130 |
| DECODE | See DECODE , page 131 |
| IFNULL | See IFNULL , page 132 |
| ISNULL | See ISNULL , page 132 |
| ISNUMERIC | See ISNUMERIC , page 133 |
| NULLIF | See NULLIF , page 134 |
| NVL | See NVL , page 134 |
| NVL2 | See NVL2 , page 136 |

COALESCE

The COALESCE function returns first value in one or more expressions that is not NULL; otherwise, it returns NULL.

Syntax

COALESCE (expression1, expression2, ...)

Remarks

COALESCE (expression1, expression2, expression3) is equivalent to this CASE statement:

```
CASE WHEN expression1 IS NOT NULL THEN expression1
      WHEN expression2 IS NOT NULL THEN expression2
      WHEN expression3 IS NOT NULL THEN expression3
      ELSE NULL END
```

The following table lists the data types of the input arguments for COALESCE, and the resulting output type.

| Data Type of expression | Output Type |
|--|--|
| BINARY, DATE, DECIMAL, FLOAT, INTEGER, INTERVAL_DAY, INTERVAL_YEAR, NULL, STRING, TIME, TIMESTAMP, XML | Follows the ANSI SQL rules for data type precedence. |

Example

```
SELECT ProductID, COALESCE (UnitPrice, SalePrice, MinPrice) "Best Price"
FROM /shared/examples/ds_orders/products products
```

DECODE

The DECODE function compares an expression with a search value and, when true, returns the specified result. If no match is found, DECODE returns the default value, if specified. If the default value is omitted, then DECODE returns NULL.

Syntax

```
DECODE (expression, search_value, result, [search_value, result]...[,default])
```

Remarks

- If the expression and search_value are NULL, the result is returned.
- To determine the data type of the output value for DECODE, using the result values, apply the ANSI SQL rules of data type precedence. The search_value has no effect on the output data type.
- DECODE treats empty strings as NULL.

The following table lists the data types of the input arguments for DECODE.

| Data Type of expression | Output Type |
|--|--|
| BINARY, DATE, DECIMAL, FLOAT, INTEGER, INTERVAL_DAY, INTERVAL_YEAR, NULL, STRING, TIME, TIMESTAMP, XML | Follows the ANSI SQL rules for data type precedence. |

Example

```
SELECT supplier_name,
```

```
DECODE (supplier_id,
10000, 'IBM',
10001, 'Microsoft',
10002, 'Hewlett Packard',
'Gateway') result
FROM suppliers;
```

This example is equivalent to:

```
CAST WHEN supplier_id = 10000 THEN 'IBM'
WHEN = 10001 THEN 'Microsoft'
WHEN = 10002 THEN 'Hewlett Packard'
ELSE 'Gateway'; END
```

IFNULL

The IFNULL function returns the value in an expression that is not NULL; otherwise, it returns a specified value.

Syntax

```
IFNULL (expression, value)
```

Remarks

The possible data types of expression must be compatible with the data type of value.

The following table lists the data types of the input arguments for IFNULL.

| Data Type of expression | Output Type |
|--|--|
| BINARY, DATE, DECIMAL, FLOAT, INTEGER, INTERVAL_DAY, INTERVAL_YEAR, NULL, STRING, TIME, TIMESTAMP, XML | Follows the ANSI SQL rules for data type precedence. |

Example

```
SELECT IFNULL (UnitPrice, 'Request Quote')
FROM /shared/examples/ds_orders/products products
```

ISNULL

The ISNULL function returns the first value in the specified expressions that is not NULL; otherwise, it returns NULL. ISNULL is equivalent to the COALESCE function except that it takes only two arguments.

Syntax

ISNULL (expression1, expression2)

Remarks

ISNULL (expression1, expression2) is equivalent to this CASE statement:

```
CASE WHEN expression1 IS NOT NULL THEN expression1
      WHEN expression2 IS NOT NULL THEN expression2
      ELSE NULL END
```

The following table lists the data types of the input arguments for ISNULL.

| Data Type of expression | Output Type |
|--|--|
| BINARY, DATE, DECIMAL, FLOAT, INTEGER, INTERVAL_YEAR, INTERVAL_DAY, NULL, STRING, TIME, TIMESTAMP, XML | Follows the ANSI SQL rules for data type precedence. |

Example

```
SELECT ProductID, ISNULL (SalePrice, UnitPrice) "Best Price"
FROM /shared/examples/ds_orders/products products
```

ISNUMERIC

The ISNUMERIC function determines whether an expression evaluates to a valid numeric type, returning 1 if it is valid and 0 if it is not valid.

Syntax

ISNUMERIC (expression)

Remarks

The following table lists the data types of the evaluated expression for ISNUMERIC and the possible return values.

| Data Type of Evaluated Expression | Returns |
|---|---------|
| CLOB, DATE, DECIMAL, FLOAT, INTEGER, INTERVAL_DAY, INTERVAL_YEAR, NULL, STRING, TIME, TIMESTAMP | 1 |
| Any other data type | 0 |

Example

```
SELECT Contact, Phone, ZipCode
WHERE ISNUMERIC (ZipCode) = 1
FROM /shared/examples/ds_orders/products products
```

NULLIF

The NULLIF function compares two arguments and returns NULL if they are equal; otherwise, it returns the first argument.

Syntax

NULLIF (expression1, expression2)

Remarks

- The first argument in NULLIF cannot be NULL. The output data type of NULLIF is always the same as the first argument.

- The function NULLIF (expression1, expression2) is equivalent to:

```
CASE
WHEN expression1 = expression2 THEN NULL
ELSE expression1
END
```

- The data types of the two input arguments must be of comparable types. The output argument data type is the same as expression1.

Example

```
SELECT ProductID, UnitPrice, NULLIF (UnitPrice, 0) as "Null Price"
FROM /shared/examples/ds_orders/products products
```

NVL

The NVL (Null Value Replacement) function tests the values returned by an expression. If the value returned is NULL, the function replaces the NULL value with the new value. If the value returned is not NULL, it is left unchanged.

Syntax

NVL (expression, new_value)

Remarks

- You can replace NULL values in a column with a value of a compatible data type.

- NVL treats empty strings as NULL. For example, NVL (nullString, '') returns NULL.
- NVL returns NULL when expression is an empty string.
- DATE and TIMESTAMP cannot be used in the same NVL command.
- NVL follows the ANSI SQL rules for data type precedence.

Example (Simple Substitution for Null Value)

```
SELECT NVL (ColumnName, 'N/A') FROM table
```

For the SELECT above, NULL values in ColumnName are replaced with the string N/A. If the input value were a column of INTEGER type, the replacement value should be an integer, and so on.

Example (Multiple NVL Function Calls)

TDV lets you issue multiple NVL function calls to replace NULL values in multiple columns. In the following example, NULL values from ColumnA are replaced with the string valueX, and NULL values from ColumnB are replaced with the value from ColumnC:

```
SELECT NVL (ColumnA, 'valueX'), NVL (ColumnB, "ColumnC") FROM table
```

The double-quotes explicitly define a column name, but the quotes can be omitted.

Example (Filtering and NVL Function Calls)

You can filter the returned result set by using the DISTINCT keyword, but it must occur outside of the NVL function call.

```
SELECT DISTINCT NVL (ColumnName, UniqueValue) FROM table
```

In the query above, all NULL values in ColumnName are replaced with UniqueValue. Because of the keyword DISTINCT, the SELECT statement returns only the first occurrence of UniqueValue.

Example (Substitution for Null Values in a Column with Values from Another Column)

Null values in one column can be replaced by the values from another column.

```
SELECT NVL (FormalTitle, Common_Name) FROM table
```

In the query above, NULL values in FormalTitle are replaced by the corresponding values from Common_Name.

NVL2

The NVL2 (Null Value Replacement 2) function lets you replace both non-NULL and NULL values in the returned result set.

Syntax

NVL2 (expression, value_if_NOT_NULL, value_if_NULL)

Remarks

- NVL2 tests the values returned by the column or variable defined by expression.
 - If a value returned is not NULL, the function replaces that value with the second expression (value_if_NOT_NULL).
 - If the value returned is NULL, the function replaces that value with the third expression (value_if_NULL).
- If a replacement value character string is not numeric or set off by single-quotes, it is interpreted as a column name. In this case, the result set is replaced with the value found in the column corresponding to the result of the NULL test.
- NVL2 treats empty strings as NULL.
- NVL2 follows the ANSI SQL rules for data type precedence.

Example (Testing for a Completion Value)

For the column named CompletionTime, a non-NULL value indicates that the transaction was completed, and so the return value is 1. If CompletionTime has a NULL value, the return value is 0.

```
NVL2 (CompletionTime, 1, 0) FROM Transaction_Table
```

Example (Checking a Timestamp)

In this example, SELECT NVL2 checks to see if a time stamp is set in the PymtPosted column. If it has a non-NULL value, the string “Yes” is returned in the result set. If the value of PymtPosted is NULL, the value from the corresponding row in the column named Acct_Status is returned in the result set.

```
SELECT NVL2 (PymtPosted_timestamp, 'Yes', Acct_Status) FROM table
```

Example (Checking for a Value or NULL)

In this example, an appropriate string is returned for each row in the named column, depending on its value.

```
SELECT NVL2 (ColName, 'This had a value.', 'This was NULL.') FROM table
```

TDV-Supported Convert Functions

Convert functions change the format of date and time values.

TDV supports the conversion functions listed in the table.

| TDV-Supported Convert Function | Comments |
|--------------------------------|---|
| CAST | See CAST , page 137 |
| FORMAT_DATE | See FORMAT_DATE , page 140 |
| PARSE_DATE | See PARSE_DATE , page 142 |
| PARSE_TIME | See PARSE_TIME , page 143 |
| PARSE_TIMESTAMP | See PARSE_TIMESTAMP , page 143 |
| TIMESTAMP | See TIMESTAMP , page 144 |
| TO_BITSTRING | See TO_BITSTRING , page 144 |
| TO_CHAR | See TO_CHAR , page 144 |
| TO_DATE | See TO_DATE , page 146 |
| TO_HEX | See TO_HEX , page 146 |
| TO_NUMBER | See TO_NUMBER , page 147 |
| TO_TIMESTAMP | See TO_TIMESTAMP , page 147 |
| TRUNC | See TRUNC (for date/time) , page 148 and TRUNC (for numbers) , page 150 |
| TRUNCATE | See TRUNCATE , page 151 |

CAST

Given a valid expression and a target data type, the CAST function converts the expression into the specified data type.

Syntax

CAST (expression AS target_data_type)

Remarks

- The expression argument specifies what is to be converted to the target data type.
- If the input expression is NULL, the output is NULL. If the input expression is an empty string, the output is an empty string. In all other cases, the output type is the same as that of the target data type.
- Target data types can include length, precision, and scale arguments.
- You can use BLOB or CLOB data types in this function.
- When you convert a DECIMAL to an INTEGER, the resulting value is truncated rather than rounded. (For example, 15.99 is converted to 15.)
- The CAST function can truncate strings without issuing an error. For example, CAST ('30000' AS INTEGER) produces an integer (30000) with no error.
- The CAST function issues a runtime error if you cast a string '30000' to TINYINT, because the TINYINT data type cannot accommodate that large a number, and no meaningful truncation can be applied. In such a case, CAST proceeds normally only if all the values of the integer column are valid values for the TINYINT data type.
- You can use the CAST function to truncate strings and round down decimals to integers.

Note: For a function to round a decimal up to the next integer, see [CEILING](#), page 184.

- All INTERVALs can be cast to CHAR and VARCHAR and vice versa.
- Interval years, months, days, hour, minute, or seconds can only be cast to identical interval units. Errors are thrown if any data loss occurs. (See examples below table.)
- CAST from character string values to DATE, TIME, or TIMESTAMP requires that the input values be in one of these ISO formats:
 - CAST to DATE—'YYYY-MM-DD' input value format
 - CAST to TIME—'HH24:MI:SS' input value format (plus optional fractional seconds with a decimal point before them)
 - CAST to TIMESTAMP—'YYYY-MM-DD HH24:MI:SS' input value format (plus optional fractional seconds with a decimal point before them)

If the values are not in these formats, you can use alternative data conversion functions such as [TO_DATE](#), page 146, [TO_TIMESTAMP](#), page 147 or [PARSE_DATE](#), page 142, [PARSE_TIMESTAMP](#), page 143, and so on. Some of these functions may not be pushed, and the query itself might not be pushed, as a result of using these functions.

The following table shows the output type that results for each combination of input expression type and target data type.

| Data Type of expression | Target Data Type | Output Type |
|--|---|-------------------|
| BIGINT, CHAR, DECIMAL, FLOAT, INTEGER, LONGVARCHAR, NUMERIC, REAL, SMALLINT, TINYINT, VARCHAR | BIGINT, DECIMAL, FLOAT, INTEGER, NUMERIC, REAL, SMALLINT, TINYINT | Target data type. |
| NULL | BIGINT, CHAR, DATE, DECIMAL, FLOAT, LONGVARCHAR, NULL, NUMERIC, INTEGER, REAL, SMALLINT, TIME, TIMESTAMP, VARCHAR | NULL |
| NULL<Data_Type1> | <Any_Data_Type2> | NULL<Data_Type1> |
| BIGINT, CHAR, DATE, DECIMAL, FLOAT, INTEGER, LONGVARCHAR, NUMERIC, REAL, SMALLINT, TIME, TIMESTAMP, TINYINT, VARCHAR | CHAR, LONGVARCHAR, VARCHAR | Target data type |
| DATE, TIMESTAMP | DATE | DATE |
| TIME, TIMESTAMP | TIME | TIME |
| BIGINT, CHAR, INTEGER, LONGVARCHAR, SMALLINT, TIMESTAMP, TINYINT, VARCHAR | TIMESTAMP | TIMESTAMP |

Example (Simple CAST Function)

```
SELECT products.UnitPrice, CAST (products.UnitPrice AS INTEGER) Price
FROM /shared/examples/ds_inventory/products products
```

Example (Target Data Type Includes Length)

```
CAST (Orders_Qry.ShipPostalCode AS CHAR(5))
```

Examples (With BLOB or CLOB)

```
CAST (myBlob AS VARBINARY(size))
```

```
CAST (myVarBinary AS BLOB)
CAST (myClob AS VARCHAR(size))
CAST (myVarChar AS CLOB)
```

Examples (Casting to Different Data Types)

```
CAST (INTERVAL '23' MONTH AS INTERVAL YEAR)
```

This returns an error (11 months lost).

```
CAST (INTERVAL '23' MONTH AS VARCHAR)
```

This returns 23 with a data type of VARCHAR.

```
CAST (INTERVAL '10' YEAR AS INTERVAL MONTH(3))
```

This returns the interval in months (120).

FORMAT_DATE

The FORMAT_DATE function formats an input argument based on a format string. The output is a VARCHAR(255).

Syntax

```
FORMAT_DATE (input, format_string)
```

Remarks

- The input argument must be a DATE, TIME, or TIMESTAMP.
- The format_string argument must be a string.
- The format_string is not case-sensitive except as indicated in the following table, which also lists the format string types.
- If input is a DATE, the format_string must not contain any TIME elements such as hour, minute, or seconds.
- If input is a TIME, the format_string must not contain any DATE elements such as year, month, or day of month.
- The output is a string representation of the DATE, TIME, or TIMESTAMP argument based on the format indicated by format_string.
- If the output exceeds 255 characters, it is truncated.

Note: Different data sources return results of FORMAT_DATE in different formats. To make sure TDV is formatting the date, put it in a CSV file and test it from that.

Any leading white space causes a parsing error. Tabs, newlines, the punctuation marks - / , . ; : and embedded or trailing white spaces are acceptable and are passed to the output. Enclose characters in single-quotes (for example, 'quoted') if you want them to be passed directly to the output. (The single-quotes are removed.) Use two single-quotes in a row to pass one single-quote to the output.

| format_string | Description |
|-------------------------|--|
| fm | Fill mode. If this is used at the start of format, excess zeroes are suppressed. |
| yyyy | 4-digit year ('2006') |
| yy | 2-digit year ('06') |
| MONTH Month month | Full month name ('JULY'). Case is matched. |
| MON Mon mon | Abbreviated month name ('JUL'). Case is matched. |
| mm | Numeric month ('07'; '7' if fill mode). |
| DAY Day day | Name of day ('FRIDAY'). Case is matched. |
| DY Dy dy | Abbreviated name of day ('FRI'). Case is matched. |
| dd | Day of month ('04'; '4' if fill mode). |
| hh | Hour in 12-hour format ('11'). |
| hh24 | Hour in 24-hour format ('23'). |
| AM am PM pm | Results are followed by AM or PM string. Case is matched. |
| mi | Minute ('59') |
| ss | Second ('59'). |
| ff | Fractional seconds to millisecond level ('790'; '79' if fill mode). |

Examples

FORMAT_DATE (DATE '2000-02-01', 'Mon mon MON Month month MONTH')

This results in: Feb feb FEB February february FEBRUARY.

FORMAT_DATE (DATE '2001-02-03', 'dd')

This results in: 03.

FORMAT_DATE (DATE '2001-02-03', 'fmdd')

This results in: 3.

FORMAT_DATE (TIME '23:59:01', 'hh hh24:mi:ss')

This results in: 11 23:59:01.

PARSE_DATE

The **PARSE_DATE** function outputs a DATE by parsing the first argument using the format defined by the second argument.

Syntax

PARSE_DATE (date_string, format_string)

Remarks

- The date_string must be a CHAR or VARCHAR.
- The format_string must also be a CHAR or VARCHAR, and must follow the same string format as the **FORMAT_DATE** function.
- The format_string must not contain any non-date elements such as hours, minutes, or seconds.
- When the two-digit year format 'yy' is used as the format string, 50 is parsed as the year 1950, but 49 is parsed as the year 2049.

Examples

PARSE_DATE ('MARCH 06, 49', 'MONTH dd, yy')

This results in a DATE value of 2049-03-06.

PARSE_DATE ('JAN 06, 2007', 'MON dd, yyyy')

This results in a DATE value of 2007-01-06.

PARSE_DATE ('MARCH 06, 50', 'MONTH dd, yy')

This results in a DATE value of 1950-03-06.

PARSE_TIME

The PARSE_TIME function is similar to [PARSE_DATE](#) except that the output of PARSE_TIME is a TIME.

Syntax

PARSE_TIME (time_string, format_string)

Remarks

The format_string must not contain any DATE elements such as year, month, or day of month.

Example

PARSE_TIME ('23:59:31', 'hh24:mi:ss')

This results in a TIME value of 23:59:31.

PARSE_TIMESTAMP

The PARSE_TIMESTAMP function is similar to [PARSE_DATE](#) except that PARSE_TIMESTAMP converts a string representing a DATE or DATETIME into a TIMESTAMP value.

Syntax

PARSE_TIMESTAMP (timestamp_string, format_string)

Examples

PARSE_TIMESTAMP ('2004-4-4 12:59:58.987654321', 'yyyy-mm-dd hh:mi:ss.ff9')

The fractional-seconds designation (ff) can be followed by an integer value from 1 to 9, indicating the number of decimal places to return.

PARSE_TIMESTAMP ('MARCH 06, 1923 03:59:31 pm', 'MONTH dd, yyyy hh:mi:ss am')

This results in a TIMESTAMP value of 1923-03-06 15:59:31.

PARSE_TIMESTAMP ('MARCH 06, 1923 23:59:31', 'MONTH dd, yyyy hh24:mi:ss')

This results in a TIMESTAMP value of 1923-03-06 23:59:31.

TIMESTAMP

The **TIMESTAMP** function converts a date or a date + time into a time stamp.

Syntax

TIMESTAMP (date_string, [time_string])

Remarks

- The date_string must be a **STRING**, **DATE**, or **DATETIME** data type.
- The time_string must be a **TIME** data type and must not contain any **DATE** elements such as year, month, or day of month.

Example

TIMESTAMP ('AUG 11, 2014')

This results in a **TIMESTAMP** value of 2014-08-11 00:00:00.

TIMESTAMP ('AUG 11, 2014', '23:59:31')

This results in a **TIMESTAMP** value of 2014-08-11 23:59:31.

TO_BITSTRING

The **TO_BITSTRING** function converts data from the binary type to the character type, where the character representation is the bitstring format.

Syntax

TO_BITSTRING (binary_expression)

Remarks

- **TO_BITSTRING** returns a **VARCHAR** that represents the given **VARBINARY** value in bitstring format.

TO_CHAR

The **TO_CHAR** function converts a date or number to a **CHAR**.

Syntax

TO_CHAR (value[, 'template'])

Remarks

- The optional template can be of any length, but make sure it contains as many digits as the longest expected input value.
- If two arguments are provided, TO_CHAR treats empty strings as NULL.
- Date templates are the same as those used in [FORMAT_DATE, page 140](#).
- Most number template indicators (commas, decimal points, letter designations) can be used in combination.
- The table below illustrates representative effects of number templates.

| Template | Sample Input | Result | Comments |
|--------------|-----------------|--------------------------------------|---|
| 999,999,999 | 12345 | 12,345 | Returns the input value with commas placed as in the template. |
| 099,999 | 1234 | 001,234 | Returns leading zeroes to fill out the number of digits in the template. |
| \$99,999 | 1234 | \$1,234 | Returns the input expressed as a dollar amount, with commas. |
| \$099,999.99 | 1234.56 1234 | \$001,234 .56 \$001,234 .00 | Returns the input expressed as a dollar amount with two decimal places, with leading zeroes to fill out the number of digits in the template. |
| L999,999 | 12345 | \$12,345 | Returns the local currency symbol in the specified position. |
| 999,999PR | -12345 | <12,345 > | If the input is negative, returns it in angle brackets. |
| s999,999 | 12345 | +12,345 | Returns the input with a leading plus or minus sign. Zero returns +0. |
| S999,999pr | -12345 | <-12,345 > | Leading S and trailing PR can be used together in the template. |

Example

```

SELECT
TO_CHAR(TIME '17:45:29', 'hh24 HH:MI:SS')
FROM
/services/databases/system/DUAL

```

This returns:

17 05:45:29

TO_DATE

The TO_DATE function converts a string value to a DATE data type.

Syntax

TO_DATE (expression, date_time_pattern)

Remarks

- The expression argument must be a CHAR or VARCHAR. For other input types, use TO_CHAR to cast a CHAR or VARCHAR before using the TO_DATE function.
- The pattern argument specifies an output pattern using a DATE, TIME, or NUMERIC format.
- You can control the data type returned by TO_DATE with a configuration parameter named Return data type of TO_DATE Function, which is under Server > SQL Engine > Overrides in the Administration > Configuration menu. If you set it to TRUE (the default), the function returns a DATE when format string is specified; if you set it to FALSE, the function returns a TIMESTAMP.
- For a change to this configuration parameter to take effect, you need to rebind or explicitly resave the view.

Example

```
SELECT TO_DATE('30 jun 2015', 'DD Mon YYYY');
```

This returns

2015-06-30

TO_HEX

The TO_HEX function converts data from the binary data type to a character data type in which the character is represented in hexadecimal format.

Syntax

TO_HEX (binary_expression)

Remarks

- The argument `binary_expression` evaluates to the integer to be converted to a hexadecimal value.
- Returns a `VARCHAR` representing the hexadecimal equivalent of a number.

Example

```
SELECT TO_HEX ('Binary'::binary(2));
```

This returns:

```
8046
```

TO_NUMBER

The `TO_NUMBER` function is deprecated. No warranties are provided to guarantee continued proper functionality. Converts a given string expression into a number.

Use the [CAST, page 137](#) function for more efficient data-type conversions.

Syntax

```
TO_NUMBER (expression)
```

The expression is a column name that returns a string, string literal, or the result of another function.

TO_TIMESTAMP

The `TO_TIMESTAMP` function is deprecated. No warranties are implied as to continued proper functionality. Converts a valid `TIMESTAMP` format into a valid `TIMESTAMP` format.

Use the [PARSE_TIMESTAMP, page 143](#) function for more efficient data-type conversions.

Syntax

```
TO_TIMESTAMP (expression)
```

The expression is a string.

TRUNC (for date/time)

The TRUNC function returns the integer portion of an expression, or, using the optional second argument, returns the expression with a specified number of decimal places. TRUNC does not take the sign of the expression into account (in other words, the decimal portion of both negative and positive expressions trend toward zero).

Syntax

TRUNC (first_arg, [format])

Remarks

- TRUNCATE works the same as TRUNC.
- The first argument is the keyword DATE or TIME or TIMESTAMP plus a quoted string containing the date or time expression to truncate.
- The data type and length of the result are the same as they are for the first argument.
- If the format argument is not present:
 - TIMESTAMP truncates to day, with a time of 00:00:00.
 - DATE or the date portion of a TIMESTAMP remains unchanged.
 - TIME or the time portion of a TIMESTAMP is returned as 00:00:00.
- The optional second argument, format, is a STRING. Its values are listed in the table below. This argument is not case-sensitive.

| Format Argument | TRUNC Output |
|---|--|
| CC SCC | Truncates to the beginning year of the century. For example, 2050-01-01 truncates to 2001-01-01. |
| SYEAR, SYYYY YEAR, YYYY, YYY, YY, Y | Truncates to the beginning of the current year. |
| IYYY, IYY, IY, I | Truncates to the beginning of the current ISO Year. An ISO year (ISO 8601 standard) starts on Monday of the week containing the first Thursday of January. It can start as early as 12/29 of the previous year, or as late as 01/04 of the current year. |
| Q | Truncates to the beginning of the current quarter. |

| Format Argument | TRUNC Output |
|--------------------|---|
| MONTH, MON, MM, RM | Truncates to the beginning of the current month. |
| WW | Same day of the current week as the first day of the year. |
| IW | Same day of the current week as the first day of the ISO year (that is, Monday). |
| W | Same day of the current week as the first day of the month. |
| DDD, DD, J | Returns the date (with 00:00:00 for the hour portion of a TIMESTAMP). |
| DAY, DY, D | Returns the date of the starting day (Sunday) of the current week. |
| IDDD | ISO day of year, where day 1 of the year is Monday of the first ISO week. Range is 001-371. |
| ID | ISO day of the week, where Monday = 1 and Sunday = 7. |
| HH, HH12, HH24 | Truncates to the hour, with 00 minutes and 00 seconds. |
| MI | Truncates to the minute, with 00 seconds. |

Examples

The table gives examples of TRUNC (or its equivalent, TRUNCATE) with its available format definitions and the results.

| SELECT Statement | Result |
|---|---------------------|
| TRUNC (TIMESTAMP '1983-03-06 12:34:56', 'cc') | 1901-01-01 00:00:00 |
| TRUNC (TIMESTAMP '1983-03-06 15:59:31', 'Y') | 1983-01-01 00:00:00 |
| TRUNC (DATE '1983-03-06', 'yyyy') | 1983-01-01 |
| TRUNC (TIMESTAMP '2015-03-06 15:59:31', 'I') | 2014-12-29 00:00:00 |
| TRUNC (DATE '2015-03-06', 'I') | 2014-12-29 |
| TRUNC (TIMESTAMP '1983-03-06 15:59:31', 'q') | 1983-01-01 00:00:00 |
| TRUNC (DATE '1983-03-06', 'q') | 1983-01-01 |

| SELECT Statement | Result |
|--|---------------------|
| TRUNC (TIMESTAMP '1983-03-06 12:34:56', 'mm') | 1983-03-01 00:00:00 |
| TRUNC (DATE '1983-03-06', 'mm') | 1983-03-01 |
| TRUNC (DATE '2015-04-03', 'ww') | 2015-04-02 |
| TRUNC (DATE '2015-04-03', 'iw') | 2015-03-30 |
| TRUNC (DATE '2015-04-03', 'w') | 2015-04-01 |
| TRUNC (TIMESTAMP '2015-04-03 12:34:56', 'ddd') | 2015-04-03 00:00:00 |
| TRUNC (TIMESTAMP '2015-04-03 12:34:56', 'd') | 2015-03-29 00:00:00 |
| TRUNC (TIMESTAMP '2015-06-10 12:34:56', 'hh') | 2015-06-10 12:00:00 |

TRUNC (for numbers)

The TRUNC function returns the integer portion of an expression, or, using the optional second argument, returns the expression with a specified number of decimal places. TRUNC does not take the sign of the expression into account; in other words, the decimal portion of both negative and positive expressions trend toward zero.

Syntax

TRUNC (expression, [decimal_places])

Remarks

The input argument expression represents the number to truncate and a NUMERIC or date/time data type as follows:

- If the first argument is a numeric expression (DECIMAL, FLOAT, INTEGER, or STRING), the second argument is the number of decimal places to truncate to.
- If the second argument is greater than the number of decimal places of the first argument, zeros are added to the right of the last significant digit.
- If the second argument is not present, the function returns the integer portion of the expression.
- The output is the same data type as the first input value.
- If either input is NULL, the output is NULL.

Examples

SELECT TRUNC(5.234);

This returns 5.

SELECT TRUNC(5.234, 2);

This returns 5.23.

SELECT TRUNC(5.234, 5);

This returns 5.23400.

TRUNCATE

The TRUNCATE function is the same as TRUNC for date/time and numeric expressions. Refer to [TRUNC \(for numbers\), page 150](#),

TRUNCATE can also be used in a SQL script to remove (“chop”) a specified number of elements from a VECTOR. Refer to [TRUNCATE, page 388](#), for a description.

TDV-Supported Cryptographic Functions

Cryptographic functions let you obfuscate product IDs, passwords, and other sensitive data.

TDV supports the cryptographic functions listed in the table.

| Cryptographic Function | Comments |
|------------------------|--|
| HASHMD2 | See HASHMD2, page 151 |
| HASHMD4 | See HASHMD4, page 152 |
| HASHSHA | See HASHSHA, page 152 |
| HASHSHA1 | See HASHSHA1, page 153 |

HASHMD2

HASHMD2 is a cryptographic hash function known as the MD2 Message-Digest Algorithm.

Syntax`HASHMD2 (value)`**Remarks**

The value argument specifies a key for use with the cryptographic algorithm; it is a STRING, BINARY, or a value that can be converted to a STRING by implicit casting. The return value is a binary hashed value.

Example`HASHMD2 (dsldkjLK85klhvn$000#knf)`

HASHMD4

HASHMD4 is a cryptographic hash function known as the MD4 Message-Digest Algorithm.

Syntax`HASHMD4 (value)`**Remarks**

The value argument specifies a key for use with the cryptographic algorithm; it is a STRING, BINARY, or a value that can be converted to a STRING by implicit casting. The return value is a binary hashed value.

Example`HASHMD4 (dsldkjLK85klhvn$000#knf)`

HASHSHA

HASHSHA is a cryptographic hash function known as the Secure Hash Function.

Syntax`HASHSHA (value)`**Remarks**

The value argument specifies a key for use with the cryptographic algorithm; it is a STRING, BINARY, or a value that can be converted to a STRING by implicit casting. The return value is a binary hashed value.

Example
HASHSHA (dsIfdkjLK85kldhmv\$*n*000#knf)

HASHSHA1

HASHSHA1 is a cryptographic hash function known as SHA-1.

Syntax
HASHSHA1 (value)

Remarks
The value argument specifies a key for use with the cryptographic algorithm; it is a STRING, BINARY, or a value that can be converted to a STRING by implicit casting. The return value is a binary hashed value.

Example
HASHSHA1 (dsIfdkjLK85kldhmv\$*n*000#knf)

TDV-Supported Date Functions

Date functions return date and time information and calculate or convert time zones.
TDV supports the date functions listed in the table.

| Date Function | Comments |
|-------------------|---|
| ADD_MONTHS | |
| AT TIME ZONE | |
| CLOCK_TIMESTAMP | |
| CURRENT_DATE | See CURRENT_DATE , page 156, |
| CURRENT_TIME | See CURRENT_TIME , page 157, |
| CURRENT_TIMESTAMP | See CURRENT_TIMESTAMP , page 157, |
| DATE_ADD | |

| Date Function | Comments |
|------------------------|--|
| DATE_PART | |
| DATE_SUB | |
| DATE_TRUNC | |
| DATEDIFF | See DATEDIFF , page 158, |
| DAY | See DAY , MONTH , and YEAR , page 159, |
| DAYOFMONTH | |
| DAYOFWEEK | |
| DAYOFWEEK_ISO | |
| DAYOFYEAR | |
| DAYS | See DAYS , page 160 |
| DAYS_BETWEEN | See DAYS_BETWEEN , page 161 |
| DBTIMEZONE | See DBTIMEZONE , page 161 |
| EXTRACT | See EXTRACT , page 162 |
| FROM_UNIXTIME | See FROM_UNIXTIME , page 163 |
| EXTRACTDAY | |
| EXTRACTDOW | |
| EXTRACTDOY | |
| EXTRACTEPOCH | |
| EXTRACTHOUR | |
| EXTRACTMICROSEC OND | |
| EXTRACTMILLISECO ND | |
| EXTRACTMINUTE | |

| Date Function | Comments |
|----------------------|--|
| EXTRACTMONTH | |
| EXTRACTQUARTER | |
| EXTRACTSECOND | |
| EXTRACTWEEK | |
| EXTRACTYEAR | |
| FROM_UNIXTIME | |
| GETUTCDATE | |
| HOUR | |
| ISFINITE | |
| JULIAN_DAY | |
| LAST_DAY | |
| MICROSECOND | |
| MIDNIGHT_SECOND S | |
| MINUTE | |
| MONTH | See DAY, MONTH, and YEAR , page 159, |
| MONTHS_BETWEEN | See MONTHS_BETWEEN , page 163, |
| NEXT_DAY | |
| NOW | |
| NUMTODSINTERVA L | See NUMTODSINTERVAL , page 164, |
| NUMTOYMINTERVA L | See NUMTOYMINTERVAL , page 164, |
| QUARTER | |
| SECOND | |

| Date Function | Comments |
|-------------------------|--|
| STATEMENT_TIMES TAMP | |
| SYSDATE | |
| TIME_SLICE | |
| TIMEOFDAY | |
| TIMESTAMP_ROUND | |
| TIMESTAMP_TRUNC | |
| TIMESTAMPADD | |
| TIMESTAMPDIFF | |
| TRANSACTION_TIMESTAMP | |
| TZ_OFFSET | See TZ_OFFSET , page 165, |
| TZCONVERTOR | See TZCONVERTOR , page 165, |
| UNIX_TIMESTAMP | |
| UTC_TO_TIMESTAMP | See UTC_TO_TIMESTAMP , page 166, |
| WEEK | |
| WEEK_ISO | |
| YEAR | See DAY, MONTH, and YEAR , page 159, |

CURRENT_DATE

The CURRENT_DATE function returns the current date from the system clock of the machine where the database is running.

Syntax
CURRENT_DATE

Remarks

- CURRENT_DATE takes no arguments.
- The output is a DATE with the format YYYY-MM-DD.

CURRENT_TIME

The CURRENT_TIME function returns the current time from the system clock of the machine where the database is running.

Syntax

CURRENT_TIME [p]

Remarks

- CURRENT_TIME has an optional precision argument (p), an unsigned integer that specifies the number of digits of fractional seconds.
- The output is a TIME with the format HH:MM:SS[.fff].
- Valid values of p are 0 (no fractional seconds) to 3 (milliseconds). Values greater than 3 return 3 digits. For example, CURRENT_TIME(3) and CURRENT_TIME(8) both return a value like 19:06:27.583.

CURRENT_TIMESTAMP

The CURRENT_TIMESTAMP function returns the current date and time from the system clock of the machine where the database is running.

Syntax

CURRENT_TIMESTAMP [p]

Remarks

- CURRENT_TIMESTAMP has an optional precision argument (p), an integer that specifies the number of digits of fractional seconds.
- The output is a TIMESTAMP with the format YYYY-MM-DD HH:MM:SS[.fff].
- Valid values of p are 0 (no fractional seconds) to 3 (milliseconds). Values greater than 3 return 3 digits. For example, CURRENT_TIMESTAMP(3) and CURRENT_TIMESTAMP(8) both return a value like 2014-12-13 13:05:47.968.

DATEDIFF

The DATEDIFF function calculates the number of date parts (days, weeks, and so on) between two specified dates, times, or timestamps.

Note: TDV supports the two parameter formats that supported data sources use. Note that the order of startdate and enddate is swapped in the two formats.

Syntax

DATEDIFF (datepart, startdate, enddate)
DATEDIFF (enddate, startdate)

Remarks

- The first argument specifies the datepart for which to return an integer indicating the difference—for example, 1 (day), 4 (years), and so on.
- TDV supports these datepart keywords:

| | | | |
|---|-------------|------|----|
| YEARS | YEAR | YYYY | YY |
| QUARTERS | QUARTER | QQ | Q |
| MONTHS | MONTH | MM | M |
| WEEKS | WEEK | WW | WK |
| WEEKS_US [an artificial date part for use in TDV only; see example 1 below] | | | |
| DAYS | DAY | DD | D |
| HOURS | HOUR | HH | |
| MINUTES | MINUTE | MI | M |
| SECONDS | SECOND | SS | S |
| MILLISECONDS | MILLISECOND | MS | |

- The other two arguments (startdate and enddate) are chronological values.
- TDV by default calculates DATEDIFF according to the ISO standard (using Monday as the first day of the week). Databases that are locale-aware (for example, Sybase) calculate according to the local standards they are configured to implement—for example, the US standard (which uses Sunday as the first day of the week). This variance in implementation can cause week-counts calculated in the data source to differ from week-counts calculated in TDV.

- WEEKS_US is an artificial datepart that makes TDV calculate DATEDIFF according to the US standard instead of the ISO standard. WEEKS_US should not be pushed to a data source, because it will be rejected there.
- Sybase produces correct (standard) results for year, month, day date parts and incorrect results for hour, minute, second date parts. TDV produces correct results for all six.

Example 1

Calculate the difference in weeks between a Friday and the following Sunday:

DATEDIFF ('WEEK', DATE '2014-04-25', DATE '2014-04-27')

According to US standard, the week starts with a Sunday; therefore, the two dates belong to different weeks (Sunday starts a new week), and so a locale-aware database produces 1.

According to ISO standard, the week starts with a Monday; therefore, Friday and Sunday belong to the same week (starting the prior Monday), so TDV produces the result 0.

If you use the artificial date part WEEKS_US, TDV produces the result 1:

DATEDIFF ('WEEKS_US', DATE '2014-04-25', DATE '2014-04-27')

Example 2

Calculate the difference in years between August 15, 2009 and December 31, 2012:

DATEDIFF ('year', date '2009-08-15', date '2012-12-31')

TDV returns 3 by counting the year intervals as follows:

[1] January 1, 2010 + [2] January 1, 2011 + [3] January 1, 2012 = 3

The months between January 1, 2012 and December 31, 2012 are ignored, because the datepart specified is YEAR, and only the start of each year is counted.

DAY, MONTH, and YEAR

The DAY, MONTH, and YEAR functions take a date expression as input, and returns the day, month, and year, respectively, from the date expression.

Syntax

DAY (date_expression)

MONTH (date_expression)

YEAR (date_expression)

Remarks

- The date_expression cannot be an empty string.
- Leading zeroes in a date or month are ignored in the output.
- If the input is NULL, the output is also NULL.

| Name and Format | Data Type of date_expression | Output Type | Output Value |
|----------------------------|------------------------------|-------------|---------------------|
| DAY (date_expression) | DATE, TIMESTAMP | INTEGER | Between 1 and 31. |
| | NULL | NULL | NULL |
| MONTH (date_expression) | DATE, TIMESTAMP | INTEGER | Between 1 and 12. |
| | NULL | NULL | NULL |
| YEAR (date_expression) | DATE, TIMESTAMP | INTEGER | Between 1 and 9999. |
| | NULL | NULL | NULL |

Example

```
SELECT DAY (orders.OrderDate) OrderDate,  
MONTH (orders.OrderDate) OrderMonth,  
YEAR (orders.OrderDate) OrderYear  
FROM /shared/examples/ds_orders/orders orders
```

DAYS

The DAYS_BETWEEN function returns the number of days since January 1, 0001, including that beginning date.

Syntax

```
DAYS (date_expression)
```

Remarks

- TDV natively implements the Vertica version of the DAYS function.
- The Excel DAYS function is far different from the TDV/Vertica DAYS function.

Examples

```
SELECT DAYS ('0001-01-02')
```

This example returns 2.

```
SELECT DAYS ('2001-01-02')
```

This example returns 730487.

DAYS_BETWEEN

The DAYS_BETWEEN function returns the number of days between two dates, excluding the two dates themselves. If the later date is first, the result is a positive number. If the earlier date is first, the result is a negative number.

The result is a NUMERIC data type.

Syntax

```
DAYS_BETWEEN (end-date, start-date)
```

Example

```
DAYS_BETWEEN ('1995-01-01', '1995-01-10')
```

This example returns a result of -9, because date1 is earlier than date2.

DBTIMEZONE

The DBTIMEZONE function returns the value of the database time zone (if the function is pushed) or the TDV time zone (if the function is not pushed).

If the function is pushed, the return type is a time-zone offset or a time-zone region name, depending on how the database time zone value was defined in the most recent CREATE DATABASE or ALTER DATABASE statement. If the function is not pushed, the return type is a time-zone offset.

Syntax

```
DBTIMEZONE
```

Example

The following example assumes that the database time zone is set to UTC time zone:

```
DBTIMEZONE ()
```

This example returns a result that looks like this:

```
DBTIME
-----
+00:00
```

EXTRACT

The EXTRACT function extracts a single field from a TIMESTAMP or INTERVAL value.

Syntax

EXTRACT (<field_name> FROM <value>)

The field_name argument is SECOND, MINUTE, HOUR, DAY, MONTH, QUARTER, or YEAR. The value argument is of type TIMESTAMP or INTERVAL.

Remarks

- The data type of the output is an exact NUMERIC with a precision equal to the leading precision of value and a scale of zero. When the field name is a SECOND, the precision is equal to the sum of the leading precision and the seconds precision of value and a scale equal to the SECOND's precision.
- When value is a negative INTERVAL, the result is a negative value.
- If value is NULL, the result is also NULL.

EXTRACT (With INTERVAL)

```
SELECT orders.OrderDate,
EXTRACT (SECOND FROM INTERVAL '2 23:51:19.124' DAY TO SECOND),
EXTRACT (MINUTE FROM INTERVAL '2 23:51:19.124' DAY TO SECOND),
EXTRACT (HOUR FROM INTERVAL '2 23:51:19.124' DAY TO SECOND),
EXTRACT (DAY FROM INTERVAL '2 23:51:19.124' DAY TO SECOND),
EXTRACT (MONTH FROM INTERVAL '500' MONTH(3))
EXTRACT (YEAR FROM INTERVAL '499-11' YEAR(3) TO MONTH),
FROM /shared/examples/ds_orders/orders
```

Results of the EXTRACT functions:

```
EXTRACT (SECOND FROM INTERVAL '2 23:51:19.124' DAY TO SECOND) = 19.124
EXTRACT (MINUTE FROM INTERVAL '2 23:51:19.124' DAY TO SECOND) = 51
EXTRACT (HOUR FROM INTERVAL '2 23:51:19.124' DAY TO SECOND) = 23
EXTRACT (DAY FROM INTERVAL '2 23:51:19.124' DAY TO SECOND) = 2
EXTRACT (MONTH FROM INTERVAL '500' MONTH(3)) = 500
EXTRACT (YEAR FROM INTERVAL '499-11' YEAR(3) TO MONTH) = 499
```

EXTRACT (Without INTERVAL)

```
SELECT orders.ShipName,
```

```
orders.OrderID,
orders.OrderDate,
EXTRACT (DAY FROM orders.OrderDate) "day",
EXTRACT (MONTH FROM orders.OrderDate) "month"
EXTRACT (QUARTER FROM orders.OrderDate) "quarter"
FROM /shared/examples/ds_orders/orders orders
```

FROM_UNIXTIME

Format a UNIX timestamp as a date.

The FROM_UNIXTIME function accepts 1 or 2 arguments. The first argument can be a date or timestamp. The second argument is a string.

Syntax

```
FROM_UNIXTIME (datetime_or_integer, [format ])
```

MONTHS_BETWEEN

The MONTHS_BETWEEN function returns the number of months between two dates.

Syntax

```
MONTHS_BETWEEN (date1, date2)
```

Remarks

- If the later date is first, the result is a positive number.
- If the earlier date is first, the result is a negative number. The number returned is also based on the real calendar.
- If the result is not a whole number of months (that is, there are some days as well), the days part is shown as a decimal (for example, 0.5 months for 15 days out of a 30-day month).
- The number is not rounded.
- Hive's MONTHS_BETWEEN rounds off the result to 8 digits decimal.
- The result is a numeric data type.

Example

```
MONTHS_BETWEEN (sysdate, TO_DATE ('01-01-2007','dd-mm-yyyy'))
```

This returns the number of months since January 1, 2007.

NUMTODSINTERVAL

The NUMTODSINTERVAL function converts a number to an INTERVAL DAY TO SECOND literal.

Syntax

NUMTODSINTERVAL (number, 'unit')

Remarks

- The number argument can be any number value, or an expression that can be implicitly converted to a number value.
- The unit argument specifies the unit-type of the number argument.
- The unit argument must be a CHAR with a value of DAY, HOUR, MINUTE, or SECOND.
- The unit argument is case-insensitive, and leading and trailing values within the parentheses are ignored.
- The precision of the return is 9.

Example

NUMTODSINTERVAL (200, ' day ')

NUMTODSINTERVAL (1200, 'Minute ')

NUMTODSINTERVAL (8, 'HOUR')

NUMTOYMINTERVAL

The NUMTOYMINTERVAL function converts a number to an INTERVAL YEAR TO MONTH literal.

Syntax

NUMTOYMINTERVAL (number, 'unit')

Remarks

- The number argument can be any number value, or an expression that can be implicitly converted to a number value.
- The unit argument specifies the unit-type of the number argument.
- The unit argument must be a CHAR with a value of YEAR or MONTH.
- The unit argument is not case-sensitive, and leading and trailing values within the parentheses are ignored.

- The precision of the return is 9.

Example

```
NUMTOYMINTERVAL (200, 'YEAR')
NUMTOYMINTERVAL (200, ' month ')
```

TZ_OFFSET

The TZ_OFFSET function returns the time zone of the argument as of the date the statement is executed. Timezone region names are required by daylight savings features.

Syntax

```
TZ_OFFSET ({ 'time_zone_name' | '{ + | - } hh : mi' })
```

Remarks

- The time_zone_name argument can be a time zone name or an offset from UTC (which returns itself).
- TDV does not accept the argument SESSIONTIMEZONE or DBTIMEZONE.
- For a list of time zone names, see [Time Zones, page 737](#)

Example

```
SELECT TZ_OFFSET ('US/Eastern');
```

This example returns a result that looks like this:

```
TZ_OFFSET('US/Eastern')
-04:00
```

TZCONVERTOR

The TZCONVERTOR function offsets a timestamp from one time zone to another time zone.

Syntax

```
TZCONVERTOR (TIMESTAMP <timestamp>, <source_zone>, <target_zone>)
```

Remarks

- The timestamp argument is in the form yyyy-mm-dd hh:mm:ss, enclosed in single-quotes.

- The `source_zone` argument is a string designating the source time zone, enclosed in single-quotes.
- The `target_zone` argument is a string designating the target time zone, enclosed in single-quotes.
- The TDV implementation of `TZCONVERTOR` does not support offset notation such as `GMT+5`.
- Valid `source_zone` / `target_zone` arguments are listed in [Time Zones, page 737](#).

Example (Date Is Outside of Daylight Saving Time Range)

```
TZCONVERTOR (TIMESTAMP '2011-3-1 00:00:00', 'US/Pacific', 'UTC')
OR
TZCONVERTOR (TIMESTAMP '2011-3-1 00:00:00', 'America/Los_Angeles', 'UTC')
```

Because daylight saving time is **not** in effect on the specified date, this example returns:

```
TIMESTAMP '2011-3-1 08:00:00'
```

Example (Date Is Inside the Daylight Saving Time Range)

```
TZCONVERTOR (TIMESTAMP '2011-9-1 00:00:00', 'US/Pacific', 'UTC')
OR
TZCONVERTOR (TIMESTAMP '2011-9-1 00:00:00', 'America/Los_Angeles', 'UTC')
```

Because daylight saving time is in effect on the specified summer date, this example returns:

```
TIMESTAMP '2011-9-1 07:00:00'
```

UTC_TO_TIMESTAMP

The `UTC_TO_TIMESTAMP` function takes a decimal or integer number—which specifies the number of seconds that have elapsed since 00:00:00 Coordinated Universal Time (UTC), Thursday, 1 January 1970—and converts it into a timestamp. Leap seconds are not counted.

The result from this function is automatically offset by the number of hours from GMT+0 of the timezone where this TDV instance resides.

Syntax

```
UTC_TO_TIMESTAMP (expression)
```

Remarks

- The expression is a DECIMAL or INTEGER specifying the number of seconds since 00:00:00 UTC.
- If the input is NULL, the result is NULL.
- The argument must not be less than -9223372036854775 or exceed 9223372036854775; otherwise, an exception occurs.

Example

UTC_TO_TIMESTAMP (36000)

This example returns a timestamp of 1970-01-01 10:00:00 if TDV Server is in time zone GMT+0, but a timestamp of 1970-01-01 02:00:00 if the TDV Server is in the America/Los_Angeles time zone (GMT-8).

TDV-Supported JSON Functions

TDV supports the JSON functions listed in the table.

| TDV-Supported JSON Function | Comments |
|-----------------------------|---|
| JSON_TABLE | See JSON_TABLE , page 167 |
| JSON_PATH | See JSONPATH , page 178 |

JSON_TABLE

JSON_TABLE is a SQL extension that creates a relational view of JSON data.

For examples of how to use JSON_TABLE with views, see the Views topic of the *TDV User Guide*. For a progressive set of JSON_TABLE examples, refer to these sections:

- [Example 1: A Literal JSON Table](#), page 170
- [Example 2: Another Literal JSON Table, with Ignored Objects](#), page 171
- [Example 3: Retrieving Object Properties and Their Values](#), page 173
- [Example 4: JSON Content Provided by an External Table](#), page 174
- [Example 5: Subquery](#), page 174
- [Example 6: Conditional Logic with Key and Value Retrieval](#), page 175

- [Example 7: Invalid Keys and Values, page 176](#)
- [Example 8: Nested Arrays, page 177](#)

Syntax

JSON_TABLE has a wide variety of arguments and syntax. After remarks, definitions, and illustrations of JSON path, the examples demonstrate how JSON_TABLE can be applied to representative use cases.

Remarks

JSON_TABLE elements can be formatted with tabs, newlines, and extra space characters to make it more readable.

With JSON_TABLE you can:

- Define and create JSON data without regard to a schema or a particular pattern of use.
- Decompose the result of JSON expression evaluation into the relational rows and columns of a new, virtual table (an “in-line relational view”).

Definitions

These definitions are most easily understood with the help of examples. Examples in this document, and more in the Views topic of the *TDV User Guide*, illustrate how JSON_TABLE can be structured, presented, and used.

- JSON—JavaScript Object Notation. No comments are allowed in this notation.
- JSON_TABLE—The keyword JSON_TABLE followed by three ordered elements, enclosed in parentheses. The first two are cross-joined either implicitly (separated by a comma) or explicitly (separated by the keywords CROSS JOIN):

- a. The JSON content provider, which can be:

A literal—A construct, enclosed in single-quotes (' '), that defines an in-line virtual table.

A column reference in an identified web data source (for example, T1, C1).

- b. A path expression (see next main bullet below), enclosed in single-quotes (' '), that designates the row provider.
- c. A COLUMNS clause—The word COLUMNS followed by, in parentheses, one or more comma-separated column definitions. Each column definition contains a column alias, its SQL data type, the keyword PATH, and either

- (1) a path expression designating the context item and object that is to occupy that column ([Example 1: A Literal JSON Table, page 170](#)), or (2) a keyword designating a syntax element whose values are to be retrieved ([Example 3: Retrieving Object Properties and Their Values, page 173](#)).
- An optional alias (for example, JT) for the table.
- If the source table is external (rather than an in-line virtual table), a comma followed by the name of the table (and an optional alias for that name).
- If the JSON content is provided through a column reference, the table that owns the column should be cross-joined with the JSON_TABLE. The tables can be cross-joined either explicitly ("T1 CROSS JOIN T2") or implicitly ("T1, T2").
- Path expression—An expression that identifies the JSON object or objects on which to operate.
 - d. Context item (JSON root)—A dollar sign (\$).
 - e. An optional path step (an object step or an array step).

Note: For column paths, a depth of only one path step is allowed (in a pattern similar to '\$.title')

- Object step—A dot (period), followed by the name of an object property. If the name includes internal dots, it must be enclosed in double quotes.
- Array step—A dot (period), followed by the name of an object property, followed by square brackets ([]). If the name includes internal dots, it must be enclosed in double quotes.

The characters inside an array step are called array slicers:

A number, or multiple numbers separated by commas, indicate the positions (counting from 1) of objects.

The keyword "to" indicates a range.

Omitting the starting number begins the range at the first element of the array.

Omitting the number after TO ends the range at the last element of the array.

Example of array steps:

[to 3, 6, 8 to] — elements 1, 2, 3, 6, 8, 9, 10 (in a 10-element array)

- Property name—In a path expression, a property name must start with an alphabetic character. It can contain alphanumerics characters and some special characters (which must be enclosed in double quotes).

JSON Paths

Here are some examples of path expressions and their meanings.

| Path Expression | Description |
|-----------------------------------|--|
| \$ | The context item (root), designating a specific JSON object. |
| \$.dept | Root, and path step. The value of property 'dept' of the object. |
| \$.dept.coffee[1] | Root, path step, and leaf step. The object that is the first element of the array that is the value of property 'coffee' of the root of the JSON object. The value of property 'coffee' is an array. |
| \$.dept.coffee[12, 3, 8 to 10] | The twelfth, third, eighth, ninth, and tenth elements of array 'coffee' (property of the root of the JSON object). The elements are returned in array order: third, eighth, ninth, tenth, twelfth. |
| \$.dept[].coffee[] | Both steps can be array steps. |
| \$. "rest.ID_output" . "rest.row" | This path expression designates a row within an external table. Notice that double quotes are used to escape the dot characters within the path elements. |

Example 1: A Literal JSON Table

This example sets up an in-line table and then selects title, author, and price (in that order) from it.

Execution results follow the query.

Query

In this example, the FROM clause provides the in-line virtual table. The JSON_TABLE literal begins right after the opening parenthesis and ends (followed by a comma) right before the path expression. The path expression specifies an array object (the virtual table) and a range from the beginning to 2. The COLUMNS clause defines columns that correspond to those requested in the SELECT. An alias of JT is applied to the table following the closing parenthesis.

```
SELECT
  myTitle, author, price
FROM
  JSON_TABLE (
    '{
      "store": {
        "book": [
          {
            "category" : "reference",
            "author" : "Nigel Rees",
```

```

        "title": "Sayings of the Century",
        "price": 8.95
    },
    { "title": "The Rumi Collection"
    },
    {
        "category": "fiction",
        "author": "Evelyn Waugh",
        "title": "Sword of Honour",
        "price": 15.00
    },
    {
        "category": "history",
        "author": "Steve Harris",
        "title": "Renaissance",
        "price": 17.00
    }
]
} }',
'$$.store.book[ to 2]'
COLUMNS (myTitle VARCHAR(100) PATH '$.title',
           price DOUBLE PATH '$.price',
           author VARCHAR(100) PATH '$.author' )) JT
ORDER BY price desc

```

Results

The results of executing this query are:

```

myTitle author price
Savings of the Century Nigel Rees 8.95
The Rumi Collection [NULL] [NULL]

```

Example 2: Another Literal JSON Table, with Ignored Objects

This example has a newsstand object between the two store objects, but the query ignores it and its contents. For every book record, the query requests the values of three attributes.

Query

```

SELECT
    myTitle, author, price
FROM
    JSON_TABLE (
        '{
            "store": {
                "book": [
                    {
                        "category": "reference",
                        "author": "Nigel Rees",
                        "title": "Sayings of the Century",
                        "price": 8.95
                    },
                    {

```

```

        "category": "fiction",
        "author": "Evelyn Waugh",
        "title": "Sword of Honour",
        "price": 15.00
      },
      {
        "category": "history",
        "author": "Steve Harris",
        "title": "Renaissance",
        "price": 17.00
      }
    ]
  },
  "newsstand": {
    "magazine": [
      {
        "brand": "Newsweek",
        "price": 10.00
      }
    ]
  },
  "store": {
    "book": [
      {
        "category": "reference",
        "author": "Nigel Rees",
        "title": "Sayings of the Century_2",
        "price": 8.95
      },
      {
        "category": "fiction",
        "author": "Evelyn Waugh",
        "title": "Sword of Honour_2",
        "price": 15.00
      },
      {
        "category": "history",
        "author": "Steve Harris",
        "title": "Renaissance_2",
        "price": 17.00
      }
    ]
  }
},
'$store[2].book'
COLUMNS (myTitle VARCHAR(100) PATH '$.title',
           price DOUBLE PATH '$.price',
           author VARCHAR(100) PATH '$.author' )) JT
-- ORDER BY price asc

```

Results

The path expression points to the second object in the array, but for that object the name test (store) does not match, so no result is returned.

Example 3: Retrieving Object Properties and Their Values

This query retrieves all of the keys and values within books. In this case, the COLUMNS clause uses keywords, instead of path expressions in single quotes, after PATH.

Query

```
SELECT
  property, propValue
FROM
  JSON_TABLE (
    '{
      "store": {
        "book": [
          {
            "category": "reference",
            "author": "Nigel Rees",
            "title": "Sayings of the Century",
            "price": 8.95
          },
          {
            "category": "fiction",
            "author": "Evelyn Waugh",
            "title": "Sword of Honour",
            "price": 15.00
          },
          {
            "category": "history",
            "author": "Steve Harris",
            "title": "Renaissance",
            "price": 17.00
          }
        ]
      }
    }',
    '$.store.book'
  )
COLUMNS (property VARCHAR(100) PATH key,
           propValue VARCHAR(200) PATH value)) JT
ORDER BY property
```

Results

The results list keys and their values as row entries, instead of listing values under column headings representing keys. In other words, you can use JSON_TABLE to retrieve structural information from tables, as well as values.

```
property propValue
author Nigel Rees
author Evelyn Waugh
author Steve Harris
category reference
category fiction
category history
price 8.95
```

```

price 15.00
price 17.00
title Savings of the Century
title Sword of Honor
title Renaissance

```

Example 4: JSON Content Provided by an External Table

This example uses `JSON_TABLE` to define a relational structure (columns) on an external table that came from a REST data source.

Query

```

SELECT
  customerId, customerName
FROM
  JSON_TABLE (
    C."output",
    '$."rest.customersResponse"."rest.customersOutput"."rest.row"'
    COLUMNS (customerId INTEGER   PATH '$."rest.customerid"',
              customerName VARCHAR(100) PATH '$."rest.companyname"')) JT ,
  /shared/customers_wrapper C

```

Results

The results are selected from the output JSON table from the REST data source.

```

customerId customerName
1Able Computing
2Anston Systems
3Blackard Electronics
...

```

Example 5: Subquery

In this example, `JSON_TABLE` is embedded in a subquery and uses a REST data source.

Query

```

SELECT
  1 C
FROM
  /services/databases/system/DUAL
WHERE EXISTS
  (
    SELECT
      customerId, price
    FROM
      /shared/examples/customers_wrapper C,
      JSON_TABLE (
        C."output",
        '$."rest.customersOutput"."rest.row"'
        COLUMNS (customerId INTEGER   PATH '$."rest.customerid"',

```

```

        price VARCHAR(100) PATH '$.rest.companyname')) JT
WHERE
    customerId = 30
)

```

Example 6: Conditional Logic with Key and Value Retrieval

This example illustrates the use of conditional logic to retrieve the value of different properties based on the structure of the source data. This adds flexibility when dealing with heterogeneous data sources.

Query

```

SELECT
    firstName,
    lastName,
    CASE WHEN firstName IS NULL THEN fullName
    ELSE firstName || ' ' || lastName END fullName,
    price
FROM
    JSON_TABLE (
        '{
            "store": {
                "book": [
                    {
                        "category": "reference",
                        "author" : {"firstName": "Nigel" , "lastName" : "Rees"},
                        "title": "Sayings of the Century",
                        "price": 8.95
                    },
                    {
                        "category": "fiction",
                        "author": {"FN":"Evelyn Waugh"},
                        "title": "Sword of Honour",
                        "price": 15.00
                    },
                    {
                        "category": "history",
                        "author": "Steve Harris",
                        "title": "Renaissance",
                        "price": 17.00
                    }
                ]
            }
        }',
        '$.store.book[1 to 2]'
        COLUMNS (author VARCHAR(100) PATH '$.author',
            price VARCHAR(100) PATH '$.price')) JT,
    JSON_TABLE (JT.author,
        '$'
        COLUMNS (firstName VARCHAR(20) PATH '$.firstName',
            lastName VARCHAR(20) PATH '$.lastName',
            fullName VARCHAR(20) PATH '$.FN' )) JT2

```

Results

The results combine data organized in two different ways, along with price, which is common to both.

| firstName | lastName | fullName | price |
|-----------|----------|--------------|-------|
| Nigel | Rees | Nigel Rees | 8.95 |
| [NULL] | [NULL] | Evelyn Waugh | 15.00 |

Example 7: Invalid Keys and Values

Query

```
SELECT
  firstName,
  lastName,
  CASE WHEN firstName IS NULL THEN author
  ELSE firstName || ' ' || lastName END fullName,
  price
FROM
  JSON_TABLE (
    '{
      "store": {
        "book": [
          {
            "category": "reference",
            "author" : {"firstName": "Nigel" , "lastName" : "Rees"},
            "title": "Sayings of the Century",
            "price": 8.95
          },
          {
            "category": "fiction",
            "author": {"FN":"Evelyn Waugh"},
            "title": "Sword of Honour",
            "price": 15.00
          },
          {
            "category": "history",
            "author": "Steve Harris",
            "title": "Renaissance",
            "price": 17.00
          }
        ]
      }
    }',
    '$.store.book[*]'
    COLUMNS (author VARCHAR(100) PATH '$.author',
              price VARCHAR(100) PATH '$.price')) JT,
  JSON_TABLE (JT.author,
    '$'
    columns (firstName VARCHAR(20) PATH '$.firstName',
            lastName VARCHAR(20) PATH '$.lastName')) JT2
```

Results

An error message is returned because the array designation (\$.store[*]) contains the wildcard character, which is not supported.

com.compositesw.cdms.webapi.WebapiException: Problems encountered while resolving JSON_TABLE references:

Exception 1 :

```
com.compositesw.cdms.services.parser.ParserException: Invalid JSON path. Cause: Compile json
path $.store.book[*] failed.. On line 32, column 6.
[parser-2931070]...
```

Example 8: Nested Arrays

In this example, store is an array that contains arrays called book. The path expression, \$.store[1].book[2], retrieves property values from these nested arrays.

Query

```
SELECT
-- {option "DISABLE_PLAN_CACHE" }
myTitle, author, price
FROM
JSON_TABLE (
'{
  "store": [{
    "book":
      [{
        "category_2": "reference",
        "author" : "Nigel Rees",
        "title": "Sayings of the Century_S1-BA1-B1",
        "price": 13.95
      } ,
      {
        "category_2": "reference",
        "author" : "Nigel Rees",
        "title": "Sayings of the Century_S1-BA1-B1",
        "price": 12.95
      }
    ],
    "book": [ {
      "category_2": "reference",
      "author" : "Nigel Rees",
      "title": "Sayings of the Century_S1-BA2-B1",
      "price": 11.95
    } ,
    {
      "category_21": "reference",
      "author" : "Nigel Rees",
      "title": "Sayings of the Century_S1-BA2-B2",
      "price": 10.95
    }
  ]
},
{
  "book": [ {
    "category_2": "reference",
```

R

ba

my

qu

my

J

pi

S

JS

Re

-

Example

```
PROCEDURE JSONPathFunctionExample(OUT resultJson VARCHAR)
BEGIN
  DECLARE sourceJson VARCHAR(4096);
  DECLARE jsonPathExpression VARCHAR(4096);

  -- Create a JSON value to use in the JSONPATH function.
  SET sourceJson = '{"LookupProductResponse":{"LookupProductResult":{"row":[{"ProductName":"Maxtific 40GB ATA133 7200","ProductID":"1","ProductDescription":"Maxtific Storage 40 GB"}]}}}';

  -- Create a JSONPATH expression to evaluate.
  SET jsonPathExpression = '$.LookupProductResponse.LookupProductResult.row[0].ProductName';

  -- Evaluate the XPATH expression against the source XML value.
  SET resultJson = JSONPATH (sourceJson, jsonPathExpression);
END
```

The result is Maxtific 40GB ATA133 7200.

TDV-Supported Numeric Functions

Numeric functions return absolute values, trigonometric values, the value of pi, and so on.

TDV supports the numeric functions listed in the table.

| Numeric Function | Comments |
|------------------|---|
| ABS | See ABS, page 181 |
| ACOS | Output value is in radians. See ACOS, page 181 |
| ASIN | Output value is in radians. See ASIN, page 182 |
| ATAN | Output value is in radians. See ATAN, page 183 |
| ATAN2 | Two-argument version of ATAN. This enables the function to use the sign of x and y to determine the quadrant of the result. See ATAN2, page 183 |
| CBRT | Returns the cubic root of a given number. |
| CEILING | See CEILING, page 184 |
| COS | Input argument is in radians. See COS, page 185 |
| COT | Input argument is in radians. See COT, page 186 |

| Numeric Function | Comments |
|------------------|--|
| DEGREES | See DEGREES, page 187 |
| EXP | See EXP, page 187 |
| FLOOR | See FLOOR, page 187 |
| LN() | Returns the natural log (base e) of a number. If you need the base 10 of a number, use the LOG function instead. |
| LOG | Returns the base 10 of a number. See LOG, page 188 If you need the base 2 (natural) number instead, use the LN () function. |
| LOG10 | Returns the log (base 10) of a number. |
| MOD | Modulo. Returns the remainder after dividing the first number by the second number. For example, 18 modulo 12 is 6 (18/12 = 1 with remainder 6, the result). |
| NUMERIC_LOG | Same as LOG. |
| Oracle ROWNUM | A number indicating the order in which Oracle selects the row from a table or set of joined rows. ROWNUM=1 for of the first row selected, ROWNUM=2 for the second row selected, and so on. |
| PI | See PI, page 188 |
| POW | Variant form of POWER. |
| POWER | See POWER, page 189 |
| RADIANS | See RADIANS, page 189 |
| RAND | Same as RANDOM. |
| RANDOM | Returns a pseudo-random FLOAT value that is greater than 0 but less than 1. |
| ROUND | See ROUND (for date/time), page 190 and ROUND (for numbers), page 193 |
| SIGN | Returns the positive or negative sign of the input expression, or 0 if the input expression resolves to zero. |
| SIN | Input argument is in radians. See SIN, page 194 |

| Numeric Function | Comments |
|------------------|--|
| SINH | See SINH , page 195 |
| SQRT | See SQRT , page 195 |
| TAN | Input argument is in radians. See TAN , page 196 |
| TANH | See TANH , page 197 |

ABS

The ABS function returns the absolute value of the input argument.

Syntax

ABS (argument)

Remarks

The table lists the valid input argument data types and the resulting output data types.

| Data Type of Argument | Output Type |
|---|---|
| BIGINT, DECIMAL, FLOAT, INTEGER, NUMERIC, REAL, SMALLINT, TINYINT | Same as the input argument. |
| NULL | NULL |
| INTERVAL | INTERVAL ABS (- INTERVAL '1' DAY) = INTERVAL '1' DAY |

Example

```
SELECT ABS(-4);
SELECT ABS(4);
```

The result in either case is 4.

ACOS

The ACOS function returns the arc-cosine of the input argument; that is, the angle (in radians) whose cosine is x.

Syntax

ACOS (x)

Remarks

The table lists the valid input argument data types and the resulting output data types.

| Data Type of Argument | Output Type | Notes |
|---|-------------|---|
| BIGINT, DECIMAL, FLOAT, INTEGER, NUMERIC, REAL, SMALLINT, TINYINT | FLOAT | Input argument is between -1.0 and +1.0. Output value is in radians. |
| NULL | NULL | |

Example

```
SELECT ACOS(0.8660254037844387)
```

The result is 0.5235987755982987 (pi/6) radians, which is 30 degrees.

ASIN

The ASIN function returns the arcsine of the input argument; that is, the angle (in radians) whose sine is x.

Syntax

ASIN (x)

Remarks

The table lists the valid input argument data types and the resulting output data types.

| Data Type of Argument | Output Type | Notes |
|---|-------------|--|
| BIGINT, DECIMAL, FLOAT, INTEGER, NUMERIC, REAL, SMALLINT, TINYINT | FLOAT | Input value is between -1.0 and +1.0. Output value is in radians. |
| NULL | NULL | |

Example

```
SELECT ASIN(0.5);
```

The result is 0.5235987755982989 radians, which is 30 degrees.

ATAN

The ATAN function returns the arctan of the input argument; that is, the angle (in radians) whose tangent is x.

Syntax

```
ATAN (x)
```

Remarks

The table lists the valid input argument data types and the resulting output data types.

| Data Type of Argument | Output Type | Notes |
|---|-------------|---|
| BIGINT, DECIMAL, FLOAT, INTEGER, NUMERIC, REAL, SMALLINT, TINYINT | FLOAT | The input value can range from -pi/2 to pi/2, inclusive. Output value is in radians. |
| NULL | NULL | |

Example

```
SELECT ATAN(0.57735026919);
```

The result is 0.5235987755982989 radians, which is 30 degrees.

ATAN2

The ATAN2 function returns the arctan value of the ratio of the input arguments; that is, the angle (in radians) whose tangent is y/x.

Syntax

```
ATAN (y, x)
```

Remarks

The table lists the valid input argument data types and the resulting output data types.

| Data Type of y and x | Output Type | Notes |
|---|-------------|---|
| BIGINT, DECIMAL, FLOAT, INTEGER, NUMERIC, REAL, SMALLINT, TINYINT | FLOAT | The input ratio y/x can range from -pi/2 to pi/2, inclusive. Output value is in radians. |
| NULL | NULL | |

Example

```
SELECT ATAN2(-5.19615242271, -9);
```

The result is 0.5773502691 radians, in the third (-x, -y) quadrant.

CEILING

The CEILING function returns the smallest integer that is greater than or equal to the input argument.

Syntax

```
CEILING (argument)
```

Remarks

The table lists the valid input argument data types and the resulting output data types.

| Data Type of Argument | Output Type |
|---|-------------|
| BIGINT, DECIMAL, FLOAT, INTEGER, NUMERIC, REAL, SMALLINT, TINYINT | INTEGER |
| NULL | NULL |

Examples

```
SELECT CEILING (3598.6);
```

The result is 3599.

```
SELECT CEILING (-3598.6);
```

The result is -3598.

COS

The COS function returns the cosine of the input argument.

Syntax

COS (argument)

Remarks

The table lists the valid input argument data types and the resulting output data types.

| Data Type of Argument | Output Type | Notes |
|---|-------------|---|
| BIGINT, DECIMAL, FLOAT, INTEGER, NUMERIC, REAL, SMALLINT, TINYINT | FLOAT | Input argument is in radians. Output value is between -1.0 and +1.0. |
| NULL | NULL | |

Example

SELECT COS(PI()/6);

The result is 0.8660254037844387.

COSH

The COSH function returns the hyperbolic cosine of the input argument.

Syntax

COSH (argument)

Remarks

The table lists the valid input argument data types and the resulting output data types.

| Data Type of Argument | Output Type | Notes |
|---|-------------|---|
| BIGINT, DECIMAL, FLOAT, INTEGER, NUMERIC, REAL, SMALLINT, TINYINT | FLOAT | Input argument is in radians. Output value range is from 1 to +infinity. |
| NULL | NULL | |

Example

```
SELECT COSH(0);
```

The result is 1.

COT

The COT function returns the cotangent of the input argument.

Syntax

COT (argument)

Remarks

The table lists the valid input argument data types and the resulting output data types.

| Data Type of Argument | Output Type | Note |
|---|-------------|-------------------------------|
| BIGINT, DECIMAL, FLOAT, INTEGER, NUMERIC, REAL, SMALLINT, TINYINT | FLOAT | Input argument is in radians. |
| NULL | NULL | |

Example

```
SELECT COT(PI()/6);
```

The result is 1.7320508075688776.

DEGREES

Given an angle in radians, the DEGREES function returns the corresponding angle in degrees.

Syntax

DEGREES (argument)

Remarks

The table lists the valid input argument data types and the resulting output data types.

| Data Type of Argument | Output Type |
|---|-------------|
| BIGINT, DECIMAL, FLOAT, INTEGER, NUMERIC, REAL, SMALLINT, TINYINT | FLOAT |
| NULL | NULL |

EXP

The EXP function returns the exponent value of the input argument.

Syntax

EXP (argument)

Remarks

The table lists the valid input argument data types and the resulting output data types.

| Data Type of Argument | Output Type |
|---|-------------|
| BIGINT, DECIMAL, FLOAT, INTEGER, NUMERIC, REAL, SMALLINT, TINYINT | FLOAT |
| NULL | NULL |

FLOOR

The FLOOR function returns the largest INTEGER that is less than or equal to the input argument.

Syntax
FLOOR (argument)

Remarks

The table lists the valid input argument data types and the resulting output data types.

| Data Type of Argument | Output Type |
|---|-------------|
| BIGINT, DECIMAL, FLOAT, INTEGER, NUMERIC, REAL, SMALLINT, TINYINT | INTEGER |
| NULL | NULL |

LOG

The LOG function returns the logarithm of the input argument.

Syntax
LOG (argument)

Remarks

The table lists the valid input argument data types and the resulting output data types.

| Data Type of Argument | Output Type | Note |
|---|-------------|--|
| BIGINT, DECIMAL, FLOAT, INTEGER, NUMERIC, REAL, SMALLINT, TINYINT | FLOAT | Input value should be greater than zero. |
| NULL | NULL | |

Example
SELECT LOG(3.1622776601683794);

The result is 0.5.

PI

The PI function returns the value of pi as a DOUBLE value.

Syntax

PI()

Remarks

The return value has 16 significant digits (3.141592653589793).

POWER

The POWER function returns the value of the first input argument raised to the power indicated by the second input argument.

Syntax

POWER (value, exponent)

Remarks

The table lists the valid input argument data types and the resulting output data types.

| Data Type of Value | Data Type of Exponent | Output Type |
|---|---|-------------|
| BIGINT, DECIMAL, FLOAT, INTEGER, NUMERIC, REAL, SMALLINT, TINYINT | BIGINT, DECIMAL, FLOAT, INTEGER, NUMERIC, REAL, SMALLINT, TINYINT | FLOAT |
| NULL | BIGINT, DECIMAL, FLOAT, INTEGER, NUMERIC, REAL, SMALLINT, TINYINT | NULL |
| BIGINT, DECIMAL, FLOAT, INTEGER, NUMERIC, REAL, SMALLINT, TINYINT | NULL | NULL |

RADIANS

Given an angle in degrees as the input argument, the RADIANS function returns the corresponding angle in radians.

Syntax

RADIANS (argument)

Remarks

The table lists the valid input argument data types and the resulting output data types.

| Data Type of Argument | Output Type |
|---|-------------|
| BIGINT, DECIMAL, FLOAT, INTEGER, NUMERIC, REAL, SMALLINT, TINYINT | FLOAT |
| NULL | NULL |

ROUND (for date/time)

Given two input arguments, this form of the ROUND function returns the value of the first input argument rounded to the value specified by the second input argument (format).

Syntax

ROUND (input_arg, format)

Remarks

- The input argument is the keyword DATE or TIME or TIMESTAMP plus a quoted string containing the date/time expression to truncate.
- If the format argument is not present:
 - TIMESTAMP rounds up or down to a day, with a time of 00:00:00.
 - DATE or the date portion of a TIMESTAMP remains unchanged.
 - TIME or the time portion of a TIMESTAMP rounds down to the given hour or up to the next hour, with 00:00 minutes and seconds.
- The optional second argument, format, is a STRING. Its values are listed in the table below. This argument is not case-sensitive.

| Format Argument | Output and Comments |
|-----------------|--|
| CC SCC | Beginning with January 1 of xx50, rounds up to the first day of the next century. Up to December 31 of xx49, rounds down to the beginning day of the current century. For example, 2050-01-01 rounds to 2101-01-01; 2049-12-31 rounds to 2001-01-01. |

| Format Argument | Output and Comments |
|---|---|
| SYEAR, SYYYY YEAR, YYYY, YYY, YY, Y | Year. Starting on July 1, rounds up to the next year. |
| IYYY, IYY, IY, I | Date of first day of the ISO year. An ISO year (ISO 8601 standard) starts on Monday of the week containing the first Thursday of January. It can start as early as 12/29 of the previous year, or as late as 01/04 of the current year. |
| Q | Date of the first day of the current quarter (up to the fifteenth of the second month of the quarter). Beginning on the sixteenth day of the second month of the quarter, rounds up to the first day of the next quarter. |
| MONTH, MON, MM, RM | Date of the first day of the current month (up to the fifteenth day). Beginning on the sixteenth day of the month, rounds up to the first day of the next month. |
| WW | Date of the same day of the week as the first day of the year. |
| IW | Because an ISO year always begins on a Monday: date of Monday of the current week if the first argument is Monday through Wednesday; date of Monday of the following week if the first argument is Thursday through Sunday. |
| W | Date of the same day of the week as the first day of the month. |
| DDD, DD, J | For 12:00:00 (noon) or later, rounds up to date of the following day. For 11:59:59 or before, or for a DATE, rounds down to current date. |
| DAY, DY, D | Starting day of the week; that is, date of the Sunday of the week that current date is in. |
| IDDD | ISO day of year, where day 1 of the year is Monday of the first ISO week. Range is 001-371. |
| ID | ISO day of the week, where Monday = 1 and Sunday = 7. |
| HH, HH12, HH24 | For hour plus 30 minutes or later, rounds up to next hour. |

| Format Argument | Output and Comments |
|-----------------|--|
| MI | For minute plus 30 seconds or later, rounds up to next minute. |

Examples

The table gives examples of ROUND with some of its format definitions and the results.

| SELECT Statement | Result |
|--|---------------------|
| ROUND (TIMESTAMP '1949-12-31 00:00:00', 'cc') | 1901-01-01 00:00:00 |
| ROUND (DATE '1950-01-01', 'cc') | 2001-01-01 |
| ROUND (timestamp '1983-07-01 15:59:31','Y') | 1984-01-01 00:00:00 |
| ROUND (date '1983-06-30', 'y') | 1983-01-01 |
| ROUND (timestamp '2015-03-06 15:59:31','i') | 2014-12-29 00:00:00 |
| ROUND (date '2015-03-06', 'i') | 2014-12-29 |
| ROUND (timestamp '1983-03-06 15:59:31','q') | 1983-01-01 00:00:00 |
| ROUND (date '1983-03-06', 'Q') | 1983-01-01 |
| ROUND (timestamp '1983-03-06 12:34:56', 'mm') | 1983-03-01 00:00:00 |
| ROUND (date '1983-03-06', 'mm') | 1983-03-01 |
| ROUND (timestamp '2015-06-08 12:34:56', 'ww') | 2015-06-11 00:00:00 |
| ROUND (date '2015-06-08', 'ww') | 2015-06-11 |
| ROUND (timestamp '2015-06-07 12:34:56', 'ww') | 2015-06-04 00:00:00 |
| ROUND (date '2015-06-107', 'ww') | 2015-06-04 |
| ROUND (timestamp '2015-06-10 12:34:56', 'ddd') | 2015-06-10 00:00:00 |
| ROUND (date '2015-06-10', 'ddd') | 2015-06-10 |
| ROUND (TIMESTAMP '2015-06-10 12:34:56', 'hh') | 2015-06-10 12:00:00 |
| ROUND (time '12:34:56', 'hh') | 12:00:00 |
| ROUND (TIMESTAMP '2015-06-10 12:34:56', 'mi') | 2015-06-10 12:34:00 |

| SELECT Statement | Result |
|-------------------------------|----------|
| ROUND (time '12:34:56', 'mi') | 12:34:00 |

ROUND (for numbers)

The ROUND function returns the value of the first input expression rounded to the number of decimal places specified by the second input argument (scale). If a third argument is present and nonzero, the input expression is truncated.

Syntax

ROUND (input_exp, scale [, modifier])

Remarks

- The input expression is the number to round.
- The input expression data type can be DECIMAL, INTEGER, FLOAT, STRING, or NULL.
- The scale data type can be DECIMAL, INTEGER, FLOAT, STRING, or NULL.
- If either the input argument or the scale is NULL, the output is NULL.
- If the modifier is present and nonzero, the input expression is truncated. If the modifier is absent or zero, the input expression is rounded. The modifier can be TINYINT, SMALLINT, or INT.
- If scale is less than zero, it is set to zero; if scale is greater than 255, it is set to 255.
- See [About SQL Functions in TDV, page 73](#) for an explanation of the DECIMAL(p,s) notation.

The table below shows the effect of scale on different input argument data types.

| Data Type of Input Argument | Output Type |
|--|---------------------------|
| DECIMAL(p,q) | DECIMAL(p-q+scale, scale) |
| TINYINT, SMALLINT, BIGINT, INTEGER, or NUMERIC | DECIMAL(19+scale, scale) |
| FLOAT, REAL, STRING | DECIMAL(255, scale) |
| NULL | NULL |

Examples

```
SELECT ROUND (columnX, 2) FROM tableY
```

If columnX is DECIMAL(10, 6), a value in columnX of 10.666666 is converted to DECIMAL(6, 2) with a value of 10.67.

```
SELECT ROUND (100.123456, 4)
```

Result is 100.1235.

```
SELECT ROUND (100.15, 4)
```

Result is 100.1500.

```
SELECT ROUND (100.15, 1, 1)
```

Because of the nonzero third argument, the result is truncated to 100.1.

SIN

The SIN function returns the sine of the input argument.

Syntax

```
SIN (argument)
```

Remarks

The table lists the valid input argument data types and the resulting output data types.

| Data Type of Argument | Output Type | Notes |
|---|-------------|--|
| BIGINT, DECIMAL, FLOAT, INTEGER, NUMERIC, REAL, SMALLINT, TINYINT | FLOAT | Input argument is in radians. Output values range from -1.0 to +1.0. |
| NULL | NULL | |

Examples

```
SELECT ROUND(SIN(PI()));
```

The result is 0.

```
SELECT SIN(PI()+0.2);
```

The result is -0.19866933079506127.

```
SELECT SIN(30 * 3.14159265359/180);
```

```
SELECT SIN(RADIANS(30));
```

The result in either case is 0.5.

SINH

The SINH function returns the hyperbolic sine of the input argument.

Syntax

SINH (argument)

Remarks

- The input argument is a double value.
- If the argument is not a number, the result is not a number.
- If the argument is zero, the result is a zero with the same sign as the argument.
- If the argument is positive infinity, the result is positive infinity.
- If the argument is negative infinity, the result is negative infinity.

Example

```
SELECT SINH(1);
```

The result is 1.17520119364.

SQRT

The SQRT function returns the square root of the input argument.

Syntax

SQRT (argument)

Remarks

The table lists the valid input argument data types and the resulting output data types.

| Data Type of Argument | Output Type | Notes |
|---|-------------|--|
| BIGINT, DECIMAL, FLOAT, INTEGER, NUMERIC, REAL, SMALLINT, TINYINT | FLOAT | Input value must not be negative. Output value is greater than or equal to 0. |
| NULL | NULL | |

Example

SELECT SQRT(6);

The result is 2.449489742783178.

TAN

The TAN function returns the tangent of the input argument.

Syntax

TAN (argument)

Remarks

The table lists the valid input argument data types and the resulting output data types.

| Data Type of Argument | Output Type | Note |
|---|-------------|-------------------------------|
| BIGINT, DECIMAL, FLOAT, INTEGER, NUMERIC, REAL, SMALLINT, TINYINT | FLOAT | Input argument is in radians. |
| NULL | NULL | |

Example

SELECT TAN(2);

The result is 0.964028.

TANH

The TANH function returns the hyperbolic tangent of the input argument.

Syntax

TANH (argument)

Remarks

- The input argument is a double value.
- If the argument is not a number, the result is not a number.
- If the argument is zero, the result is a zero with the same sign as the argument.
- If the argument is positive infinity, the result is +1.0.
- If the argument is negative infinity, the result is -1.0.

Example

SELECT TANH(1);

The result is 0.76159415595.

TDV-Supported Operator Functions

TDV supports the operator functions listed in the table.

| Operator Function | Comments |
|-------------------|--|
| X + Y | Add |
| X Y | Concatenate; for example abc def returns abcdef. |
| X/Y | Divide; for example, 18/3 returns 6. |
| X ** Y | Exponentiate; for example, 2**8 returns 256. |
| FACTORIAL or X! | Return the factorial of the given integer; for example, 5! returns 60. |
| X % Y | Modulo; for example 7 % 3 returns 1, because seven divided by 3 leaves a remainder of 1. |
| X * Y | Multiply. |

| Operator Function | Comments |
|-------------------|--|
| -X | Negate (unary operator); for example, -(1) returns -1 and -(-1) returns 1. |
| X - Y | Subtract. |

TDV-Supported Phonetic Functions

TDV supports the phonetic functions listed in the table. The TDV functions are modeled on Netezza implementations. For further information, follow [this link](#).

| Phonetic Function | Comments |
|-------------------|--|
| DBL_MP | DBL_MP (string_expression) returns a TDV 32-bit numeric expression of the input argument. |
| NYSIIS | NYSIIS (string_expression) returns a Soundex representation of the input argument using the New York State Identification and Intelligence System (NYSIIS) variation of Soundex. |
| PRI_MP | PRI_MP (numeric_expression) returns the four-character primary metaphone string from the numeric_expression returned by DBL_MP. |
| SCORE_MP | SCORE_MP (numeric_expression1, numeric_expression2) returns a score for how closely the two numeric expressions match. |
| SEC_MP | SEC_MP (numeric_expression) returns the four-character secondary metaphone string from the numeric_expression returned by DBL_MP. |

TDV-Supported Utility Function

TDV supports a utility function named EXPLAIN. This function makes the query execution plan available to JDBC clients (as well as Studio users). The actual query is not executed.

| Option | Description | Example Syntax |
|-------------------------|--|--|
| show_source_plan="true" | Retrieves the query plan. This can also be used in the SQL Scratchpad. | <pre>explain select {option show_source_plan="true"} * from <view></pre> |

| Option | Description | Example Syntax |
|---------------------|--|---|
| show_runtime="true" | Retrieves the execution statistics (plan and runtime statistics). This can also be used in the SQL Scratchpad. | explain select {option show_runtime="true"} * from <view> |

Syntax

EXPLAIN <any_SQL-statement>

Remarks

Preceding any SQL statement with the keyword EXPLAIN makes the query execution plan available in a text format that can be displayed either in Studio or in a JDBC client.

TDV-Supported XML Functions

TDV supports a number of functions that apply to XML content.

As part of generating a valid XML element name, characters that are not allowed in XML are escaped.

The following sections provide information about escaping:

- [Identifier Escaping, page 200](#)
- [Text Escaping, page 201](#)

TDV supports the XML functions listed in the table.

| XML Function | Comments |
|---------------|---|
| XMLAGG | See XMLAGG, page 100 (where it is grouped with other aggregate functions) |
| XMLATTRIBUTES | See XMLATTRIBUTES, page 201 |
| XMLCOMMENT | See XMLCOMMENT, page 202 |
| XMLCONCAT | See XMLCONCAT, page 202 |
| XMLDOCUMENT | See XMLDOCUMENT, page 203 |
| XMLELEMENT | See XMLELEMENT, page 203 |

| XML Function | Comments |
|---------------|--|
| XMLFOREST | See XMLFOREST , page 204 |
| XMLNAMESPACES | See XMLNAMESPACES , page 205 |
| XMLPI | See XMLPI , page 205 |
| XMLQUERY | See XMLQUERY , page 205 |
| XMLTEXT | See XMLTEXT , page 207 |
| XPATH | See XPATH , page 207 |
| XSLT | See XSLT , page 208 |

Note: The following functions are part of the ANSI specification but not supported in TDV: XMLTABLE, XMLITERATE, XMLBINARY, XMLCAST, XMLEXISTS, XMLPARSE, XMLSERIALIZE, XMLVALIDATE.

Identifier Escaping

When creating XML nodes with XML elements, the name of the node can be escaped according to ANSI specification 9075-14, paragraph 4.10.3. The ANSI specification provides two modes of escaping:

- full escaping
- partial escaping

TDV Server uses partial escaping. Only alphabetical characters and underscore can be leading characters. All other characters are converted.

Partially escaped identifiers escape all nonleading numerical characters except minus (-), underscore (_), and colon (:), with the format `_xDDDD_` where DDDD is the hexadecimal equivalent of the ASCII character. For example, the ampersand character (&) is converted to `_x0026_`.

Examples

`XMLELEMENT (NAME "29", 'text')`

This results in `<_x0032_9>text</_x0032_9>`
`XMLFOREST ('black' AS "a:")`

This results in `<_x003A_>black</_x003A_>`
`XMLFOREST ('black' AS "a:-")`

This results in <a:->black<a:->

Text Escaping

In an XML text, characters are replaced as listed in the following table.

| Character in an XML Function | Replacement |
|------------------------------|-------------|
| & | & |
| > | > |
| < | < |
| " | " |
| ' | ' |

Examples

XMLTEXT ('&')

The replacement results in &
XMLFOREST ('>' AS green)

The replacement results in <green>></green>
XMLELEMENT (NAME red, "")

The replacement results in <red>"</red>

XMLATTRIBUTES

The XMLATTRIBUTES function constructs XML attributes from the arguments provided. The result is an XML sequence with an attribute node for each input value.

Syntax

XMLATTRIBUTES (<XML_attribute_value> [AS <XML attribute_name>] [{ , <XML_attribute_value> [AS <XML attribute_name>] }...])

In the syntax, XML_attribute_value is a value expression, and XML_attribute_name is the element identifier.

Remarks

- XMLATTRIBUTES can only be used as an argument of the XMLELEMENT function.
- This function requires the AS keyword if aliases are used. This is in contrast to the select-list, which does not require the AS keyword for aliasing.
- This function cannot be used to insert blank spaces or newline characters.
- Any <value expression> that evaluates to NULL is ignored.
- Each <value expression> must have a unique attribute name.
- If the result of every <value expression> is NULL, the result is NULL.

Example

```
SELECT XMLELEMENT (name Details, XMLATTRIBUTES (product_id,name as "Name"),
XMLELEMENT (name orderno, OrderID),
XMLELEMENT (name status, Status),
XMLELEMENT (name price, UnitPrice)) myOutput
FROM /shared/examples/ds_orders/orderdetails
WHERE ProductID < 20
```

XMLCOMMENT

The XMLCOMMENT function generates an XML comment based on a value expression.

Syntax

```
XMLCOMMENT (value_expression)
```

Remarks

- The instruction argument is a string designating the processing instruction to generate.
- The value_expression argument must resolve to a string.
- The value returned takes the form <!--string-->.

XMLCONCAT

The XMLCONCAT function concatenates one or more XML fragments.

Syntax

```
XMLCONCAT ( <XML value expression> { , <XML value expression> } ...
[ <XML returning clause> ] )
```

Remarks

- If an argument evaluates to NULL, that argument is ignored.
- If all arguments are NULL, the result is NULL.
- If only one non-NULL argument is supplied, the result of the function is that argument.

Example

```
SELECT XMLCONCAT (XMLTEXT (customers.ContactFirstName), XMLTEXT (' '),
XMLTEXT (customers.ContactLastName)) AS CustomerName
FROM /shared/examples/ds_orders/customers customers
```

XMLDOCUMENT

The XMLDOCUMENT function generates an XML value with a single XQuery document node. It is equivalent to running the XQUERY expression.

Syntax

```
XMLDOCUMENT ( <XML_value_expression> [ <XML_returning_clause> ] )
```

The <XML_value_expression> is a sequence of nodes of atomic values.

Example

```
SELECT XMLDOCUMENT (XMLELEMENT (name Details, XMLATTRIBUTES (ProductID as product),
XMLELEMENT (name orderno, OrderID),
XMLELEMENT (name status, Status),
XMLELEMENT (name price, UnitPrice))) myXMLDocument
FROM /shared/examples/ds_orders/orderdetails
WHERE ProductID < 20
```

XMLELEMENT

The XMLELEMENT function creates an XML node with an optional XML attributes node.

Syntax

```
XMLELEMENT ( NAME <XML_element_name>
[ , <XML_namespace_declaration> ] [ , <XML_attributes> ]
[ { , <XML_element_content> } ...
[ OPTION <XML_content_option> ] ]
[ <XML_returning_clause> ] )
```

Remarks

- The first argument, XML_element_name, is the name of the XML node. It can be escaped if it contains certain characters. For details, see [Identifier Escaping, page 200](#).
- The optional second argument, XML_namespace_declaration, is the XMLNAMESPACE function.
- The optional third argument, XML_attributes, is the XMLATTRIBUTES function.
- The optional fourth argument, XML_element_content, is the content of the XML node, which can be an XML, numeric, or character type.
- If XML_element_content evaluates to a character literal, it is escaped. For details, see [Text Escaping, page 201](#).

Example

```
SELECT XMLELEMENT (name Details, XMLATTRIBUTES (ProductID AS product),
XMLELEMENT (name orderno, OrderID),
XMLELEMENT (name status, Status),
XMLELEMENT (name price, UnitPrice)) myOutput
FROM /shared/examples/ds_orders/orderdetails
WHERE ProductID < 20
```

XMLFOREST

The XMLFOREST function creates a series of XML nodes, with the arguments being the children of each node. XMLFOREST accepts one or more arguments.

Syntax

```
XMLFOREST ( [ <XML_namespace_declaration> . ] <forest_element_list>
[ OPTION <XML_content_option> ]
[ <XML_returning_clause> ]
)
```

Remarks

- Each argument to XMLFOREST can be followed by an optional alias. The alias becomes the name of the XML node and the argument becomes a child of that node.
- If no alias is specified and the argument is a column, the name of the column is the name of the XML node.
- If an argument is not a column, an error is generated.
- If an argument evaluates to a character literal, the resulting string is escaped.

Example

```
SELECT XMLFOREST (CompanyName AS name, City AS city) AS
NameAndCityOfCompany
FROM /shared/examples/ds_orders/customers
```

XMLNAMESPACES

XMLNAMESPACES constructs namespace declarations from the arguments provided. Namespaces provide a way to distinguish names used in XML documents.

A namespace declaration can only be used as an argument for specific functions such as XMLELEMENT and XMLFOREST. The result is one or more XML namespace declarations containing in-scope namespaces for each non-NULL input value.

Example

```
SELECT CustomerID, XMLELEMENT (NAME customerName,
XMLNAMESPACES ('http://localhost:9400/services/webservices/ws/TestService/TestPort' AS "customers"),
XMLATTRIBUTES (City AS city, ContactLastName as name)) "Customer Details"
FROM /services/webservices/ws/TestService/TestPort/customers
WHERE StateOrProvince = 'CA'
```

XMLPI

The XMLPI function generates an XML processing instruction node and adds it to an XML element being constructed with [XMLELEMENT](#), [page 203](#).

Syntax

XMLPI (instruction [, expression])

Remarks

- The instruction argument is a string designating the processing instruction to generate.
- The string_expression argument returns a value of a built-in character or graphic string.

XMLQUERY

The XMLQUERY function returns an XML value from the evaluation of an XQuery expression. This function accepts one character literal argument, which is the XML query.

Syntax

```
XMLQUERY ( <XQuery_expression> [ <XML_query_argument list> ]
[ <XML_returning_clause>
[ <XML_query_returning_mechanism> ] ]
<XML_query_empty_handling_option>
)
```

Remarks

- Multiple arguments can be passed as input to the XML query.
- Each argument must be an XML data type, or be castable to an XML data type.
- Each argument can be followed by an optional identifier which gives the argument a variable name.
- If an argument is missing the identifier, the argument becomes the context item.
- Only one context item per XMLQUERY function can exist.
- Each input must be resolved to an XML data type and must be aliased.
- Each alias must be unique, and is case-sensitive.
- TDV Server uses the Saxon as its XQuery parser. Saxon requires that all XQuery variables be declared as external variables in the XQuery. (This is not an ANSI requirement.)
- TDV Server also requires all noncontext item variables to be declared in the XQUERY text. (This is not ANSI-specific.)
- Variables can be declared through the format declare variable \$<name> external; where <name> is the name of the variable. Multiple declarations can be separated by a semicolon.
- XQuery keywords should be written in lowercase.
- The XML-passing mechanism is accepted but ignored.

If the empty handling option is NULL ON EMPTY, NULL is returned if the result of the XQuery is an empty element.

Example

```
XMLQUERY ('declare variable $c external; for $i in $c
where $i /PDName = "Jean Morgan"
order by $i/PDName
return $i/PDName' passing XMLELEMENT(name PDRecord, XMLELEMENT(name PDName, 'Jean Morgan')) as c
)
```

This results in <PDName>Jean\ Morgan</PDName>.

XMLTEXT

The XMLTEXT function returns an XML value having the input argument as its content. XMLTEXT accepts a character argument and returns the string after it has been escaped. See section [Text Escaping, page 201](#)

Syntax

XMLTEXT (<character_value_expression> [<XML_returning_clause>])

Remark

- If the character argument evaluates to NULL, NULL is returned.
- The character value expression can accept NULL, INTEGER, FLOAT, DECIMAL, DATE, TIMESTAMP, TIME, CLOB, BLOB, VARCHAR, and CHAR.

Example

```
SELECT XMLELEMENT (name company,
XMLTEXT (customers.CompanyName) ) "Company Name", XMLTEXT (customers.City) City
FROM /shared/examples/ds_orders/customers customers
```

XPATH

The XPATH function uses path expressions to navigate to nodes in an XML document.

Syntax

XPATH (sourceXml, xpathExpression)

Remarks

- The first argument is the name of an XML document.
- The second argument is a string value containing an XPATH expression.
- The function evaluates the XPATH expression against the supplied XML value and returns the results as an XML value.

Example

```
PROCEDURE XpathFunctionExample (OUT resultXml XML)
BEGIN
DECLARE sourceXml XML;
DECLARE xpathExpression VARCHAR(4096);
-- Create an XML value to use in the XPATH function.
SET sourceXml = '<Book><Chapter>Test Data</Chapter></Book>';
-- Create an XPATH expression to evaluate.
```

```

SET xpathExpression = '//Chapter';
-- Evaluate the XPATH expression against the source XML value.
SET resultXml = XPATH (sourceXml, xpathExpression);
END

```

XSLT

The XSLT function creates a new XML document based on the content of a source XML document. XSLT can be used to convert data from one XML schema to another, or to convert XML data into web pages or PDF documents.

Syntax

XSLT (sourceXml, xsltExpression)

Remarks

- The first argument is the name of an XML document.
- The second argument is a string value containing an XSLT expression.
- The function evaluates the XSLT expression against the supplied XML value and returns the results as an XML value.

Note: For further information, refer to the open-source Saxon XSLT home page, <http://saxon.sourceforge.net/>.

Example

```

PROCEDURE XsltFunctionExample (OUT resultXml XML)
BEGIN
  DECLARE sourceXml XML;
  DECLARE xsltExpression VARCHAR(4096);
  -- Create an XML value to use in the XSLT function.
  SET sourceXml =
    '<Book><Chapter>Test Data</Chapter></Book>';
  -- Create an XSLT expression to evaluate.
  SET xsltExpression =
    '<?xml:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
      <xsl:output omit-xml-declaration="true"/>
      <xsl:strip-space elements="*" />
      <xsl:template match="/">
        <itemA>
          <xsl:for-each select="/Book">
            <itemB>
              <xsl:value-of select="Chapter"/>
            </itemB>
          </xsl:for-each>
        </itemA>
      </xsl:template>
    </xsl:stylesheet>';
  -- Evaluate the XSLT expression against the source XML value.
  SET resultXml = XSLT (sourceXml, xsltExpression);

```

END

TDV Support for SQL Operators

TDV supports several types of operators that you can add to SQL statements to perform arithmetic operations, compare values, combine them, or check for certain conditions. This topic describes these operators, shows their syntax, lists their input and output data types and gives examples of their use.

The arithmetic operators are built-in. For example, you can select them from a drop-down list (Function > Operator) for a Column on a Grid panel.

You must manually type comparison, logical, and condition operators into a query on a SQL or SQL Script panel.

TDV supports the following types of SQL operators:

- [Arithmetic Operators, page 211](#)
- [Comparison Operators, page 230](#)
- [Logical Operators, page 232](#)
- [Condition Operators, page 234](#)

Arithmetic Operators

The following arithmetic operators are built-in. You can select them from a cell drop-down list on a Grid panel:

- [Add, page 212](#)
- [Concatenation, page 217](#)
- [Divide, page 217](#)
- [Exponentiate, page 219](#)
- [Factorial, page 219](#)
- [Modulo, page 219](#)
- [Multiply, page 220](#)
- [Negate, page 224](#)
- [Subtract, page 225](#)

The table below summarizes the operator names and their symbols.

| Operator Name | Symbol | String or Symbol Name |
|---------------|--------|--------------------------------|
| Add | + | Plus sign |
| Concatenate | | Double-pipe; two vertical bars |
| Divide | / | Forward slash |
| Exponentiate | ** | Double-asterisk |
| Factorial | ! | Exclamation mark |
| Factorial | | “FACTORIAL” |
| Modulo | % | Percent sign |
| Multiply | * | Asterisk |
| Negate | - | Hyphen (minus sign) |
| Subtract | - | Hyphen (minus sign) |

Add

The add operator (+) adds two operands and returns the sum.

Note: A configuration parameter is available to control whether this operator allows precision/scale to exceed 38. See [Decimal Digit Limitation on Functions, page 523](#), for details.

DECIMAL and NUMERIC Data Types

When the add operator is applied to operands that include DECIMAL or NUMERIC data types, the output data type, precision and scale might depend on the data type, precision and scale of the operands, as shown below.

Syntax

operand1 + operand2

Remarks

- The order of the inputs (operands) has no effect on the output data type.

- The outputs for DECIMAL and NUMERIC data types combined with other operands are shown in the table.

| Inputs | Output |
|---------------------------------------|---|
| DECIMAL(p1,s1) + DECIMAL(p2,s2) | DECIMAL(p3,s3), with p3 the larger precision of the inputs plus 1, and s3 the larger scale of the inputs. |
| DECIMAL(p1,s1) + NUMERIC | |
| NUMERIC + NUMERIC | NUMERIC |
| DECIMAL(p,s) + not-DECIMAL-or-NUMERIC | DECIMAL(p,s) |
| NUMERIC + not-DECIMAL-or-NUMERIC | NUMERIC |

Example

DECIMAL(6,1) + NUMERIC(4,2) -> DECIMAL(7,2)

INTERVAL Type

INTERVAL can be added to DATE, TIME, TIMESTAMP or another INTERVAL.

Syntax

operand1 + operand2

Remarks

- INTERVAL days, hours, minutes, or seconds can only be added to other INTERVAL days, hours, minutes, or seconds. INTERVAL years or months can only be added to other INTERVAL years or months. The two groups of units are not interchangeable.
- When adding months, the TDV Server does not round down the day of the month, and it might throw an error if the day of the month is invalid for the specified month.
- The order of the inputs (operands) has no effect on the output data type.
- The outputs for INTERVAL added to various operands are shown in the table.

| Inputs | Output |
|---------------------|----------|
| INTERVAL + INTERVAL | INTERVAL |

| Inputs | Output |
|--|--|
| INTERVAL + DATE DATE + INTERVAL | DATE. Only days, months, and years can be added to a DATE. |
| INTERVAL + TIME TIME + INTERVAL | TIME |
| INTERVAL + TIMESTAMP TIMESTAMP + INTERVAL | TIMESTAMP |

Examples

```
DATE '1999-12-31' + INTERVAL '1' DAY = DATE '2000-01-01'  
INTERVAL '1' MONTH + DATE '1999-12-31' = DATE '2000-01-31'  
DATE '1989-03-15' + INTERVAL '1' YEAR = DATE '1990-03-15'  
DATE '2000-01-31' + INTERVAL '1' MONTH = <Error: February only has 28 days>  
INTERVAL '6000' SECOND(4) + INTERVAL '3000' DAY(4) = INTERVAL '3000 01:40:00' DAY(4) TO SECOND  
INTERVAL '6000' SECOND(4) + TIME '7:00:00' = TIME '08:40:00'
```

Mixed Data Types

The add operator can be applied to operands that have a wide variety of data types, including operands comparable or castable to data types that can accept arithmetic operators.

Syntax

```
operand1 + operand2
```

Remarks

The operand data types and resulting output data types are shown in the table.

| Operand1 Type | Operand2 Type | Output Type |
|--|--|-------------|
| TINYINT SMALLINT INTEGER BIGINT | TINYINT SMALLINT INTEGER BIGINT STRING | INTEGER |
| TINYINT SMALLINT INTEGER BIGINT | FLOAT REAL | FLOAT |

| Operand1 Type | Operand2 Type | Output Type |
|--|--|-------------|
| TINYINT SMALLINT INTEGER BIGINT | DECIMAL NUMERIC | DECIMAL |
| TINYINT SMALLINT INTEGER BIGINT STRING | DATE | DATE |
| TINYINT SMALLINT INTEGER BIGINT STRING | TIMESTAMP | TIMESTAMP |
| FLOAT REAL | TINYINT SMALLINT INTEGER BIGINT STRING | FLOAT |
| FLOAT REAL | FLOAT REAL | |
| FLOAT REAL | DECIMAL NUMERIC | DECIMAL |
| FLOAT REAL | DATE | DATE |
| FLOAT REAL | TIMESTAMP | TIMESTAMP |
| DECIMAL NUMERIC | TINYINT SMALLINT INTEGER BIGINT | DECIMAL |
| DECIMAL NUMERIC | FLOAT REAL | |

| Operand1 Type | Operand2 Type | Output Type |
|--------------------|--|--|
| DECIMAL NUMERIC | DECIMAL NUMERIC | FLOAT |
| DECIMAL NUMERIC | DATE | DATE |
| DECIMAL NUMERIC | TIMESTAMP | TIMESTAMP |
| DATE | INTERVAL | DATE |
| DATE | STRING | DATE |
| TIMESTAMP | INTERVAL | TIMESTAMP |
| TIMESTAMP | STRING | TIMESTAMP |
| STRING | STRING TINYINT SMALLINT INTEGER BIGINT NUMERIC FLOAT REAL DECIMAL DATE TIMESTAMP | DECIMAL INTEGER INTEGER INTEGER INTEGER DECIMAL FLOAT FLOAT DECIMAL DATE TIMESTAMP |
| NULL | TINYINT SMALLINT INTEGER BIGINT NUMERIC FLOAT REAL DECIMAL DATE TIMESTAMP NULL | NULL |

| Operand1 Type | Operand2 Type | Output Type |
|--|---------------------------------------|---------------------------------------|
| TINYINT SMALLINT INTEGER BIGINT NUMERIC FLOAT REAL DECIMAL DATE TIMESTAMP STRING | NULL | NULL |
| INTERVAL | DATE INTERVAL TIME TIMESTAMP | DATE INTERVAL TIME TIMESTAMP |

Concatenation

The concatenation operator (||) concatenates the first operand and second operand and returns the combined operands.

Syntax

operand1 || operand2

Example

abc || def

This concatenation returns abcdef.

Divide

The divide operator (/) divides the first operand by the second and returns the quotient.

Note: A configuration parameter is available to control whether this operator allows precision/scale to exceed 38. See [Decimal Digit Limitation on Functions, page 523](#), for details.

DECIMAL and NUMERIC Data Types

When the divide operator is applied to operands that include DECIMAL or NUMERIC data types, the output data type, precision and scale might depend on the data type, precision and scale of the operands, as shown below.

Syntax

operand1 / operand2

Remarks

- The order of the inputs (operands) has no effect on the output data type.
- The outputs for dividing DECIMAL and NUMERIC data types are shown in the table.

| operand1 | operand2 | Output |
|----------------|----------------|-------------------------|
| DECIMAL(p1,s1) | DECIMAL(p2,s2) | DECIMAL(p1+p2+s2,s1+p2) |
| DECIMAL(p,s) | NUMERIC | DECIMAL(p,s) |
| NUMERIC | NUMERIC | NUMERIC |

If the input is DECIMAL or NUMERIC with any number data types other than DECIMAL or NUMERIC, the output data type should be DECIMAL or NUMERIC, respectively, with the same precision and scale as the DECIMAL or NUMERIC input.

Example

DECIMAL(12,3) / DECIMAL(45,2)

This division operation returns DECIMAL(59,48).

INTERVAL Type

INTERVAL can be divided by numbers. The output is an INTERVAL.

Syntax

INTERVAL / NUMERIC

Example

INTERVAL '90' HOUR / 10 = INTERVAL '0 09:00:00' DAY TO SECOND
INTERVAL '1' YEAR / .1 = INTERVAL '10-00' YEAR TO MONTH

Exponentiate

Exponentiation (**) combines a number and an exponent. For example, 2**3 takes the number 2 to the exponent 3 and returns two cubed, or 8.

Syntax

number ** exponent

Example

10**4

This expression returns 10 to the fourth power, or 1000.

Factorial

Factorial is an operator (!) and a function (FACTORIAL) that returns the factorial product of an integer.

Note: Twenty-factorial (20! or 2.432902e+18) is the largest factorial product that TDV natively supports. It is 9.223372e+18, which is within the range of BIGINT (-2**63 to +2**63 - 1). For maximum values in pushed functions, refer to the appropriate section of [Function Support for Data Sources, page 519](#)

Syntax

operand !
FACTORIAL(n)

Examples

FACTORIAL(5)
5!

Both of these return 120 (1 * 2 * 3 * 4 * 5).

Modulo

The modulo operator (%) divides the first operand by the second operand (the modulus) and returns the remainder.

Note: A configuration parameter is available to control whether this operator allows precision/scale to exceed 38. See [Decimal Digit Limitation on Functions, page 523](#), for details.

Syntax

operand1 % operand2

Example

11 % 3

Eleven modulo 3 is 2; that is, 11 divided by 3 has a remainder of 2.

Remarks

The input (operand1 and operand2) data types and resulting output data types are shown in the table.

| Operand1 | Operand2 | Output |
|--|--|---------|
| TINYINT SMALLINT INTEGER BIGINT STRING | TINYINT SMALLINT INTEGER BIGINT STRING | INTEGER |
| NULL | TINYINT SMALLINT INTEGER BIGINT STRING | NULL |
| TINYINT SMALLINT INTEGER BIGINT | NULL | NULL |

Multiply

The multiply operator (*) multiplies two operands and returns the product.

Note: A configuration parameter is available to control whether this operator allows precision/scale to exceed 38. See [Decimal Digit Limitation on Functions, page 523](#), for details.

DECIMAL and NUMERIC Data Types

When the multiply operator is applied to operands that include DECIMAL or NUMERIC data types, the output data type, precision and scale might depend on the data type, precision and scale of the operands, as shown below.

Syntax

operand1 * operand2

Remarks

- The order of the inputs (operands) has no effect on the output data type.
- The outputs for multiplying DECIMAL and NUMERIC data types with each other and with other data types are shown in the table.

| Inputs | Output |
|--|----------------------|
| DECIMAL(p1,s1) * DECIMAL(p2,s2) | DECIMAL(p1+p2,s1+s2) |
| DECIMAL(p1,s1) * NUMERIC(p2,s2) | |
| NUMERIC(p1,s1) * NUMERIC(p2,s2) | NUMERIC(p1+p2,s1+s2) |
| DECIMAL(p1,s1) * TINYINT | DECIMAL(p+3,s) |
| DECIMAL(p1,s1) * SMALLINT | DECIMAL(p+5,s) |
| DECIMAL(p1,s1) * INTEGER | DECIMAL(p+10,s) |
| DECIMAL(p1,s1) * BIGINT | DECIMAL(p+19,s) |
| DECIMAL(p,s) * not-DECIMAL-or-NUMERIC | DECIMAL(p,s) |
| NUMERIC(p,s) * not-DECIMAL-or-NUMERIC | NUMERIC(p,s) |

Examples

DECIMAL(6,2) * TINYINT -> DECIMAL(9,2)

DECIMAL(6,2) * SMALLINT -> DECIMAL(11,2)

INTERVAL Type

INTERVAL can be multiplied by numbers. The output data type is INTERVAL.

Syntax
INTERVAL * NUMERIC

Examples
INTERVAL '1' DAY * 10 = INTERVAL '10 00:00:00' DAY TO SECOND
INTERVAL '10' DAY * .1 = INTERVAL '1 00:00:00' DAY TO SECOND

Mixed Data Types

The multiply operator can be applied to operands that have a wide variety of data types, including operands comparable or castable to data types that can accept arithmetic operators.

Syntax
operand1 * operand2

Remarks
The operand data types and resulting output data types are shown in the table.

| Operand1 | Operand2 | Output |
|--|--|---------|
| TINYINT SMALLINT INTEGER BIGINT | TINYINT SMALLINT INTEGER BIGINT STRING | INTEGER |
| TINYINT SMALLINT INTEGER BIGINT | FLOAT REAL | FLOAT |
| TINYINT SMALLINT INTEGER BIGINT | DECIMAL NUMERIC | DECIMAL |

| Operand1 | Operand2 | Output |
|--------------------|---|---|
| FLOAT REAL | TINYINT SMALLINT INTEGER BIGINT DECIMAL | FLOAT |
| FLOAT REAL | FLOAT REAL | |
| FLOAT REAL | DECIMAL NUMERIC | DECIMAL |
| DECIMAL NUMERIC | TINYINT SMALLINT INTEGER BIGINT STRING | |
| DECIMAL NUMERIC | FLOAT REAL | |
| DECIMAL NUMERIC | DECIMAL NUMERIC | FLOAT |
| STRING | STRING TINYINT SMALLINT INTEGER BIGINT NUMERIC FLOAT REAL DECIMAL | DECIMAL INTEGER INTEGER INTEGER INTEGER DECIMAL FLOAT FLOAT DECIMAL |

| Operand1 | Operand2 | Output |
|---|---|----------|
| NULL | TINYINT SMALLINT INTEGER BIGINT NUMERIC FLOAT REAL DECIMAL STRING NULL | NULL |
| TINYINT SMALLINT INTEGER BIGINT NUMERIC FLOAT REAL DECIMAL STRING | NULL | |
| INTERVAL | NUMERIC | INTERVAL |

Negate

The negate operator (-) returns the negative value of an operand. Negate is a unary operator: it acts on a single operand.

INTERVAL Type

INTERVAL can be negated in various ways, as shown in the following examples:

```
- INTERVAL '1' DAY
INTERVAL '-1' DAY
INTERVAL -'1' DAY
```

Other Data Types

Negate can be applied to the following data types: BIGINT, DECIMAL, FLOAT, INTEGER, INTERVAL, NULL, NUMERIC, REAL, SMALLINT, STRING, and TINYINT.

Negate does not change the operand’s data type.

Subtract

The subtract operator (-) subtracts the second operand from the first operand and returns the difference.

Note: A configuration parameter is available to control whether this operator allows precision/scale to exceed 38. See [Decimal Digit Limitation on Functions, page 523](#), for details.

DECIMAL and NUMERIC Data Types

When the subtract operator is applied to operands that include DECIMAL or NUMERIC data types, the output data type, precision and scale might depend on the data type, precision and scale of the operands, as shown below.

Syntax

operand1 - operand2

Remarks

- The order of the inputs (operands) has no effect on the output data type.
- The outputs for DECIMAL and NUMERIC data types combined with other operands are shown in the table.

| Inputs | Output |
|---------------------------------------|--|
| DECIMAL(p1,s1) - DECIMAL(p2,s2) | DECIMAL(p3,s3), with p3 the larger precision of the inputs, and s3 the larger scale of the inputs. |
| DECIMAL(p1,s1) - NUMERIC(p2,s2) | |
| NUMERIC - NUMERIC | NUMERIC |
| DECIMAL(p,s) - not-DECIMAL-or-NUMERIC | DECIMAL(p,s) |
| NUMERIC - not-DECIMAL-or-NUMERIC | NUMERIC |

Examples

DECIMAL(6,1) - DECIMAL(5,2) -> DECIMAL(6,2)
DECIMAL(6,1) - NUMERIC(5,2) -> DECIMAL(6,2)
NUMERIC(6,1) - NUMERIC(5,2) -> NUMERIC(6,2)

INTERVAL Type

INTERVAL can be subtracted from DATE, TIME, TIMESTAMP or another INTERVAL.

Syntax

operand1 - operand2

Remarks

- INTERVAL can be subtracted from DATE, TIME, TIMESTAMP, or another INTERVAL.
- Interval days, hours, minutes, or seconds can only be subtracted from other interval days, hours, minutes, or seconds. Interval years or months can only be subtracted from other interval years or months. The two groups of units are not interchangeable.
- When subtracting months, the TDV Server does not round down the day of the month, and it might throw an error if the day of the month is invalid for the specified month.
- The order of the inputs (operands) has no effect on the output data type.
- The outputs for INTERVAL as a subtract operand are shown in the table.

| Inputs | Output |
|----------------------|--|
| DATE - INTERVAL | DATE. Only days, months, and years can be subtracted from a DATE. |
| INTERVAL - INTERVAL | INTERVAL |
| INTERVAL - DATE | DATE. Dates can be subtracted from INTERVALs only if the INTERVAL is days, months, or years. |
| INTERVAL - TIME | TIME |
| INTERVAL - TIMESTAMP | TIMESTAMP |

Examples

TIME '7:00:00' - INTERVAL '0 3:00:00' DAY TO SECOND = TIME '4:00:00'
INTERVAL '10000-11' YEAR(5) TO MONTH - INTERVAL '1' MONTH(1) = INTERVAL '10000-10' YEAR TO MONTH
DATE '1999-12-31' - INTERVAL '365' DAY(3) = DATE '1998-01-01'

Mixed Data Types

The subtract operator can be applied to operands that have a wide variety of data types, including operands comparable or castable to data types that can accept arithmetic operators.

Syntax

operand1 - operand2

Remarks

The operand data types and resulting output data types are shown in the table.

| Operand1 | Operand2 | Output |
|--|--------------------------------|---------------|
| TINYINT | TINYINT | INTEGER |
| SMALLINT | SMALLINT | |
| INTEGER | INTEGER | |
| BIGINT | BIGINT | |
| TINYINT SMALLINT INTEGER BIGINT | STRING | INTEGER |
| TINYINT SMALLINT INTEGER BIGINT | FLOAT REAL | FLOAT |
| TINYINT SMALLINT INTEGER BIGINT | DECIMAL (p,s) NUMERIC (p,s) | DECIMAL (p,s) |

| Operand1 | Operand2 | Output |
|--------------------|--|---|
| Float Real | TINYINT SMALLINT INTEGER BIGINT | Float |
| Float Real | Float Real | |
| Float | DECIMAL (p,s) | |
| Real | DECIMAL (p,s) NUMERIC (p,s) | DECIMAL |
| DECIMAL NUMERIC | TINYINT SMALLINT INTEGER BIGINT STRING | |
| DECIMAL NUMERIC | Float Real | DECIMAL |
| DECIMAL NUMERIC | DECIMAL NUMERIC | DECIMAL |
| DATE | DATE | An INTERVAL day: the number of days between the two arguments. DATE '2006-03-20' - DATE '2005-12-02' = INTERVAL '108' DAY(3) |
| DATE | TIMESTAMP STRING | An INTEGER that represents the difference between the dates in the two inputs. |
| TIME | TIME | An INTERVAL hour to second. TIME '21:00:00' - TIME '19:00:00' = INTERVAL '0 2:00:00' DAY TO SECOND |
| TIMESTAMP | TIMESTAMP | An INTERVAL day to second. TIMESTAMP '2006-03-20 21:00:00' - TIMESTAMP '2005-12-02 19:00:00' = INTERVAL '108 02:00:00' DAY(3) TO SECOND |

| Operand1 | Operand2 | Output |
|--|----------------|--|
| TIMESTAMP | DATE STRING | An INTEGER that represents the difference between the dates in the two inputs. |
| STRING | STRING | DECIMAL |
| | TINYINT | INTEGER |
| | SMALLINT | INTEGER |
| | INTEGER | INTEGER |
| | BIGINT | INTEGER |
| | NUMERIC | DECIMAL |
| | FLOAT | FLOAT |
| | REAL | FLOAT |
| | DECIMAL | DECIMAL |
| | DATE | INTEGER |
| | TIMESTAMP | INTEGER |
| NULL | TINYINT | NULL |
| | SMALLINT | |
| | INTEGER | |
| | BIGINT | |
| | NUMERIC | |
| | FLOAT | |
| | REAL | |
| | DECIMAL | |
| | DATE | |
| | TIMESTAMP | |
| | STRING | |
| TINYINT SMALLINT INTEGER BIGINT NUMERIC FLOAT REAL DECIMAL DATE TIMESTAMP STRING | NULL | |

Comparison Operators

TDV supports the following comparison operators:

- = (equal to)
- <> (not equal to)
- < (less than)
- > (greater than)
- <= (less than or equal to)
- >= (greater than or equal to)

These operators are **not** available through the Studio interface, so you must manually type them into a query on a **SQL** or **SQL Script** panel.

If the value of the operand on either side of the comparison operator is NULL, the output of the logical comparison is also NULL. In the examples below, any row with a ProductID value of NULL does not return a result.

Example (Equal To)

```
SELECT ProductName, UnitPrice
FROM /shared/examples/ds_inventory/products products
WHERE ProductID = 5
```

Example (Not Equal To)

```
SELECT ProductName, UnitPrice
FROM /shared/examples/ds_inventory/products products
WHERE ProductID <> 10
```

Example (Less Than)

```
SELECT ProductName, UnitPrice
FROM /shared/examples/ds_inventory/products products
WHERE ProductID < 10
```

Example (Greater Than)

```
SELECT ProductName, UnitPrice
FROM /shared/examples/ds_inventory/products products
WHERE ProductID > 10
```

Example (Less Than Or Equal To)

```
SELECT ProductName, UnitPrice
FROM /shared/examples/ds_inventory/products products
WHERE ProductID <= 5
```

Example (Greater Than Or Equal To)

```
SELECT ProductName, UnitPrice
FROM /shared/examples/ds_inventory/products products
WHERE ProductID >= 5
```

Quantified Comparisons

When a comparison operator is used together with the words ALL, ANY, or SOME, the comparison is known as being “quantified.” Such comparisons operate on subqueries that could return multiple rows but would return a single column.

Syntax

```
<expression> <comparison-operator> {ALL |ANY |SOME} <column-subquery>
```

Remarks

- <comparison-operator> can be <, =, >, <=, >=, <>.
- ALL or ANY is applicable only to subqueries. When one of them is used, the comparison converts a scalar subquery to a column subquery.
- Except for use in subqueries, ANY and SOME are equivalent.
- If ALL is used, the comparison must be true for all values returned by the subquery.
- If ANY or SOME is used, the comparison must be true for at least one value of the subquery.
- A subquery using ANY must return a single column. ANY compares a single value to the column of data values produced by the subquery.

If any of the comparisons yields a value of TRUE, the ANY comparison returns TRUE. If the subquery returns NULL, the ANY comparison returns FALSE.

- ALL is used to compare a single value to the data values produced by the subquery. The specified comparison operator is used to compare the given value to each data value in the result set. If all of the comparisons returns a value of TRUE, the ALL test also returns TRUE.
- If the subquery returns an empty result set, the ALL test returns a value of TRUE.

If the comparison test is false for any values in the result set, the ALL search returns FALSE.

The ALL search returns TRUE if all the values are true. Otherwise, it returns UNKNOWN. For example, if there is a NULL value in the subquery result set but the search condition is TRUE for all non-null values, the ALL test returns UNKNOWN.

- Negating an ALL comparison is not equivalent to using an ALL comparison with any other combination of operators. For example, NOT a = ALL (subquery) is not equivalent to a <> ALL (subquery).

Example (Using ANY)

This query returns the order ID and customer ID for orders placed after at least one product with an order ID of 500 was shipped.

```
SELECT ID, CustomerID
FROM SalesOrders
WHERE OrderDate > ANY (
    SELECT ShipDate
    FROM SalesOrderItems
    WHERE ID=500);
```

Example (Using SOME)

You can use SOME instead of ANY, as in the following example:

```
SELECT ID, CustomerID
FROM SalesOrders
WHERE OrderDate > SOME (
    SELECT ShipDate
    FROM SalesOrderItems
    WHERE ID=500);
```

Example (Using ALL)

The main query tests the order dates for each order against the shipping dates of every product with the ID 500. If an order date is greater than the shipping date for every shipment with order ID 500, the ID and customer ID from the SalesOrders table are included in the result set.

```
SELECT ID, CustomerID
FROM SalesOrders
WHERE OrderDate > ALL (
    SELECT ShipDate
    FROM SalesOrderItems
    WHERE ID=500);
```

Logical Operators

TDV supports three logical operators:

- [AND, page 233](#)
- [NOT, page 233](#)
- [OR, page 234](#)

AND

AND returns rows that must satisfy all of the given conditions.

Syntax

condition1 AND condition2

Remark

This operator is **not** available through the Studio interface, so you must manually type it into a query on a **SQL** or **SQL Script** panel.

Example

```
SELECT ProductID, ProductName, ProductDescription
FROM /shared/examples/ds_inventory/products products
WHERE ReorderLevel > 5 AND LeadTime = '1 Day'
```

NOT

NOT returns rows that do not satisfy a condition.

Syntax

NOT expression

NOT expression1 AND NOT expression2

Remarks

- This operator is **not** available through the Studio interface, so you must manually type it into a query on a **SQL** or **SQL Script** panel.
- The expressions can be fixed values or comparisons.

Example (Single NOT)

```
SELECT orderdetails.*
FROM /shared/examples/ds_orders/orderdetails orderdetails
WHERE NOT (UnitPrice > 100.00)
```

Example (Two NOTs)

```
SELECT orderdetails.*
```

```
FROM /shared/examples/ds_orders/orderdetails orderdetails
WHERE NOT (UnitPrice > 100.00) AND NOT (Quantity < 2)
```

OR

OR returns rows that must satisfy at least one of the given conditions.

Syntax

```
condition1 OR condition2
```

Remarks

- This operator is **not** available through the Studio interface, so you must manually type it into a query on a SQL or SQL Script panel.

Example

```
SELECT ProductID, ProductName, ProductDescription
FROM /shared/examples/ds_inventory/products products
WHERE ReorderLevel > 5 OR UnitPrice > 22.00
```

Condition Operators

TDV supports the following condition operators:

- [CASE, page 235](#)
- [COALESCE, page 236](#)
- [DECODE, page 237](#)
- [IN and NOT IN, page 239](#)
- [IS NOT NULL, page 241](#)
- [IS NULL, page 242](#)
- [LIKE, page 242](#)
- [OVERLAPS, page 243](#)

These operators are **not** available through the Studio interface, so you must manually type them into a query on a SQL or SQL Script panel.

CASE

The CASE operator is used to evaluate several conditions and return a single value for the first matched condition. The CASE expression is similar to an IF-THEN-ELSE or a SWITCH statement used in many programming languages. However, in SQL, CASE is an expression, not a statement.

CASE has two formats:

- [Simple CASE, page 235](#)
- [Searched CASE, page 236](#)

Simple CASE

A simple CASE compares an expression to a set of simple expressions.

Syntax

```
CASE <comparison-value>
WHEN <conditional-expression 1> THEN <scalar-expression 1>
WHEN <conditional-expression 2> THEN <scalar-expression 2>
WHEN <conditional-expression 3> THEN <scalar-expression 3>
[ELSE <default-scalar-expression>]
END
```

Remarks

- Using CASE, you can express an alternate value to an underlying value. For example, if the underlying value is a code (such as 1, 2, 3), you can display it as a humanly readable string value (Small, Medium, Large), without affecting the underlying value.
- If none of the test conditions is true, CASE returns the result contained in the optional ELSE case, if one is specified.
- If no match is found and ELSE is not specified, ELSE NULL is assumed by default.

Example

```
SELECT ProductID, Status, UnitPrice,
CASE Status
WHEN 'open' THEN UnitPrice * 1.10
WHEN 'closed' THEN UnitPrice * 1
ELSE UnitPrice
END
AS "New Price"
FROM /shared/examples/ds_orders/orderdetails
```

Searched CASE

A searched CASE compares an expression to a set of logical expressions.

Syntax

```
CASE
WHEN <conditional_expression_1> THEN <scalar_expression_1>
WHEN <conditional_expression_2> THEN <scalar_expression_2>
WHEN <conditional_expression_3> THEN <scalar_expression_3>
[ELSE <default_scalar_expression>]
END
```

Examples

```
SELECT ProductID, UnitPrice
CASE
WHEN UnitPrice <=100 THEN 'Between $1 and $100.00'
WHEN UnitPrice <=200 THEN 'Between $100.01 and $200.00'
ELSE 'Over $200.00'
END
AS "Price Range"
FROM /shared/examples/ds_orders/orderdetails
```

```
SELECT ProductID, UnitPrice
CASE
WHEN UnitPrice > 400 THEN 'Above 400.00'
WHEN UnitPrice >=300 THEN 'Between 300 and 400.00'
END
AS "Price Range"
FROM /shared/examples/ds_orders/orderdetails
```

COALESCE

COALESCE returns the first non-null expression among its arguments.

Syntax

```
COALESCE (expression1, expression2, expression3...)
```

This is equivalent to:

```
CASE
WHEN expression1 NOT NULL THEN expression1
WHEN expression2 NOT NULL THEN expression2
ELSE expression3
END
```


Remarks

TDV Server supports push of the COALESCE functional expression directly to the following data sources to take advantage of any indices that might yield a performance advantage: DB2, MySQL, Netezza, Oracle, SQL Server, Sybase, and Teradata.

Example

```
SELECT
CAST (COALESCE (hourly_wage * 40 * 52, salary, commission * num_sales) AS money)
FROM wages
```

DECODE

DECODE allows data value transformation during run-time retrieval.

Syntax

DECODE (expression, string1, result1 [, stringN, resultN][, default]) columnNameAlias

Remarks

The DECODE function is similar to an IF-THEN-ELSE statement, where a regular expression can be compared to one or more values, and if the expression equals a specified value, the corresponding replacement value is returned.

- DECODE can be used to resolve strings into digital values for counting or other purposes.
- The expression and any of the strings can be a table.column, a regular expression, or values that are compared with each other for equality.
- The expression must resolve to a single value, but the string can be any value that resolves to TRUE or FALSE in an equality function.
- If the compared arguments are equal, the value of the result corresponding to the string is returned; otherwise, the specified default value or null is returned.
- Each string is compared with the expression in sequential order, even if the expression does not match a prior string.
- If a default value is specified, it is returned if the expression does not match any of the strings.

Example (Expanding a One-Letter Code)

This example performs a mapping from a one-letter code to a more meaningful value.

```
SELECT TBL_user.user_id "User ID",
DECODE (TBL_user.gender,
'F', 'Female',
'M', 'Male',
'unspecified') Gender,
TBL_user.first_name "First Name"
FROM /shared/examples/NORTHBAY/"user" TBL_user
```

Similar syntax could be used to convert a pair of one-letter Boolean values (T/F, 1/0, etc.) to a value of TRUE or FALSE.

Example (Mapping States to Regions)

This example performs a mapping from states to regions.

```
SELECT *,
DECODE (customers.StateOrProvince,
'Al', 'East',
'Ak', 'North',
'Ar', 'Midwest',
'Az', 'West',
'Somewhere else') Region
FROM /shared/examples/ds_orders/customers customers
ORDER BY Region
```

Example (Nesting DECODE in Other Functions)

DECODE can be nested within other functions. This can be useful for counting occurrences of a particular value.

In this example, the number of suppliers in each of three states is counted after deriving a string to either a 1 or a 0.

```
SELECT
SUM (DECODE (suppliers.StateOrProvince, 'CA', 1, 0)) California,
SUM (DECODE (suppliers.StateOrProvince, 'NY', 1, 0)) "New York",
SUM (DECODE (suppliers.StateOrProvince, 'PA', 1, 0)) Pennsylvania
FROM /shared/examples/ds_inventory/suppliers
```

EXISTS and NOT EXISTS

The EXISTS keyword tests the existence of specific rows in the result of a subquery. The NOT EXISTS keyword tests for the nonexistence of specific rows in the result of a subquery.

Syntax (EXISTS)

```
<source-expression>
```

WHERE EXISTS <subquery>

Syntax (NOT EXISTS)

```
<source-expression>
WHERE NOT EXISTS <subquery>
```

Remarks

- EXISTS checks for the existence of rows under conditions specified in the subquery; the actual values in the rows are irrelevant. Therefore, the SELECT clause in the subquery is SELECT * to retrieve all columns.
- The subquery can return any number of rows and columns.
- The subquery returns at least one row if the EXISTS condition is met and the NOT EXISTS condition is false.
- If the subquery does not return any rows, the EXISTS condition is not met and the NOT EXISTS condition is true.
- Even if the rows returned by the subquery contain NULL values, they are not ignored. Such rows are considered normal rows.

Example (EXISTS)

```
SELECT *
FROM /shared/examples/ds_inventory/suppliers
WHERE EXISTS (SELECT *
  FROM /shared/examples/ds_inventory/purchaseorders
  WHERE purchaseorders.SupplierID = 5)
```

Example (NOT EXISTS)

```
SELECT *
FROM /shared/examples/ds_inventory/suppliers
WHERE NOT EXISTS (SELECT *
  FROM /shared/examples/ds_inventory/purchaseorders
  WHERE purchaseorders.SupplierID = 100)
```

IN and NOT IN

The IN operator is used to determine whether a given value matches any value in a list of target values. The list of target values can be generated using a subquery.

The IN operator has two formats. One format uses an expression; the other uses a subquery.

Syntax 1

```
<source-expression [ , source-expression]>
[NOT] IN <scalar-expression-list>
```

Syntax 2

```
<source-expression [, source-expression]>
[NOT] IN <subquery [, subquery]>
```

Remarks

- IN is a comparison operator like < (less than) or LIKE.
- IN is valid anywhere a conditional expression can be used. That is, you can place IN in a WHERE clause, a HAVING clause, or a JOIN ON clause, as well as in a CASE expression.
- All the expressions in the target list (<scalar-expression-list>) must be compatible or implicitly castable to the source expression (<source-expression>), or vice versa.
- If the items in the target list are not all of the same type, as in the following example:

```
ID IN (1000, 'X', 12.0)
```

the list is translated to the following format:

```
(left = right1) OR (left = right2) OR (left = right3)
```

with CASE functions as necessary.

- You can use IN with data types that are comparable or implicitly castable to each other.
- You can combine IN conditions with AND and OR conditions.
- The expression A IN (B, C) is equivalent to the expression A = B or A = C.
- You can use NOT IN to negate the IN condition. That is, NOT IN specifies values that are not in the target list.
- The subquery can return only one column of a compatible data type. However, it can return multiple rows.
- The subquery is run once prior to running the parent query, to populate the list of values for the IN clause.
- You can combine IN conditions using AND and OR conditions.
- IN can take multiple source (left-side) expressions, and multiple values in the subquery. However, the number of values on the right side must match the number of values on the left side.
- Multiple sets of values are allowed.

Example (Syntax 1, Using IN with a String)

```
SELECT customers.CompanyName, customers.StateOrProvince
```

```
FROM /shared/examples/ds_orders/customers customers
WHERE StateOrProvince IN ('CA', 'PA')
```

Example (Syntax 1, Using IN with a Number)

```
SELECT ProductId, ProductName
FROM /shared/examples/ds_inventory/products
WHERE CategoryID IN (5,6)
```

Example (Syntax 1, Using IN with Date)

```
SELECT purchaseorders.ShipDate, SupplierID
FROM /shared/examples/ds_inventory/purchaseorders PurchaseOrders
WHERE ShipDate IN (CAST ('2003-02-06' AS DATE), CAST ('2003-02-07' AS DATE) )
```

Example (Syntax 1, Using IN with AND and OR)

```
SELECT purchaseorders.ShipDate, SupplierID
FROM /shared/examples/ds_inventory/purchaseorders PurchaseOrders
WHERE ShipDate IN (TO_DATE ('2003-02-06'))
AND ShippingMethodID = 3
OR DatePromised = '2003-02-02'
OR ShipDate IN ('2001-05-08', DATE '2001-04-01', '2000-02-25')
```

Example (Syntax 2, Using IN)

```
SELECT Customers.ContactName
FROM /shared/examples/ds_orders/Customers Customers
WHERE City IN (SELECT City
                FROM /shared/examples/ds_orders/Customers Customers
                WHERE City = 'New York')
```

Example (Syntax 2, Using NOT IN)

```
SELECT Customers.ContactName, CompanyName
FROM /shared/examples/ds_orders/Customers Customers
WHERE City
NOT IN (SELECT City
        FROM /shared/examples/ds_orders/Customers Customers
        WHERE City = 'New York')
```

IS NOT NULL

The IS NOT NULL operator matches a non-null value.

Syntax

```
WHERE x IS NOT NULL
```

Example

```
SELECT Employees.FirstName, Employees.LastName, Employees.WorkPhone
FROM /services/databases/ds_service/Employees Employees
WHERE BillingRate IS NOT NULL
```

IS NULL

The IS NULL operator matches a null value.

Syntax

```
WHERE x IS NULL
```

Example

```
SELECT Employees.FirstName, Employees.LastName, Employees.WorkPhone  
FROM /services/databases/ds_service/Employees Employees  
WHERE BillingRate IS NULL
```

LIKE

The LIKE operator is used to match strings based on a pattern.

Syntax

```
column LIKE pattern [ESCAPE escape-character]
```

Remarks

The pattern string can contain wild-card characters that have special meaning:

- % (percent sign). Matches any sequence of zero or more characters.
- _ (underscore). Matches any single character.

Example (Like with Percent-Sign Match)

```
SELECT ProductID, ProductName, ProductDescription  
FROM /shared/examples/ds_inventory/products products  
WHERE ProductName LIKE 'Acme%'
```

The pattern matches Acme Memory, Acme Processor, and Acme Storage 40GB.

Example (Like with Underscore Match)

```
SELECT company, credit_limit  
FROM customers  
WHERE company LIKE 'Smiths_n'
```

The pattern matches Smithson and Smithsen, but not Smithsonian.

If the data value in the column is null, the LIKE test returns a NULL result.

You can locate strings that do not match a pattern by using NOT LIKE.

Example (Using The ESCAPE Character)

The ESCAPE character is used to match the wild-card characters themselves, as shown here.

```
SELECT order_num, product
FROM orders
WHERE product LIKE 'A$%BC%' ESCAPE '$'
```

The first percent sign is not treated as wild-card character, because it is preceded by the \$ escape character.

OVERLAPS

The OVERLAPS operator returns TRUE when two time periods (defined by their endpoints) overlap, FALSE when they do not overlap.

Syntax

```
(start1, end1) OVERLAPS (start2, end2)
(start1, length1) OVERLAPS (start2, length2)
```

Remarks

- The endpoints can be specified as pairs of dates, times, or time stamps; or as a date, time, or time stamp followed by an interval.
- When a pair of values is provided, either the start or the end can be written first. OVERLAPS automatically takes the earlier value of the pair as the start.
- Each time period is considered to represent the half-open interval start <= time < end, unless start and end are equal, in which case it represents that single time instant. This means, for instance, that two time periods with only an endpoint in common do not overlap.

Examples

```
SELECT (DATE '2016-04-16', DATE '2016-11-25') OVERLAPS
      (DATE '2016-11-28', DATE '2017-11-28');
```

The result is TRUE.

```
SELECT (DATE '2016-02-16', INTERVAL '120 days') OVERLAPS
      (DATE '2016-11-28', DATE '2017-11-28');
```

The result is FALSE.

```
SELECT (DATE '2016-09-29', DATE '2016-11-28') OVERLAPS
      (DATE '2016-11-28', DATE '2016-11-29');
```

The result is FALSE.

```
SELECT (DATE '2016-05-05', DATE '2016-05-05') OVERLAPS  
      (DATE '2016-05-05', DATE '2016-05-05');
```

The result is TRUE.

TDV Query Engine Options

This topic describes the TDV SQL query engine hints (options) used to suggest how the execution plan might be optimized.

Execution of SQL views, procedures, and transactions created with TDV-defined resources follows an optimized execution plan. The execution plan is generated dynamically based on how the SQL is written, what and how native resources are being used, TDV configuration settings, the presence of data-source-specific statistical data, and the presence of TDV SQL query engine options.

The following apply to this topic:

- Keywords (option names and values) are not case-sensitive. For example, "TRUE" and "true" are equivalent. However, in this documentation, they are presented in all-uppercase.
- If a TRUE/FALSE option is specified without a value, it is implicitly set to TRUE. For example, the syntax definition `CASE_SENSITIVE[={"TRUE" | "FALSE"}]` means that you can specify `CASE_SENSITIVE` (with no value) or `CASE_SENSITIVE="TRUE"` to set it to TRUE, or specify `CASE_SENSITIVE="FALSE"` to set it to FALSE.

Query engine options let the developer influence the generation of the execution plan by overriding, for specific SQL statements and keywords, TDV configuration settings. The configuration settings can be found in Studio by navigating to the parameters under TDV Server > SQL Engine.

- [DATA_SHIP_MODE Values, page 246](#)
- [GROUP BY Options, page 247](#)
- [INSERT, UPDATE, and DELETE Options, page 249](#)
- [JOIN Options, page 251](#)
- [ORDER BY Options, page 258](#)
- [SELECT Options, page 259](#)
- [UNION, INTERSECT, and EXCEPT Options, page 271](#)

DATA_SHIP_MODE Values

DATA_SHIP_MODE is a SELECT option that controls automatic rework of federated queries across data sources. Reworked table selections can be shipped through an API to temporary tables so that query nodes can be joined with local tables.

DATA_SHIP_MODE modifies how the query engine handles queries that are candidates for data ship optimization.

When any of these DATA_SHIP_MODE options is specified in a query, it overrides the value specified in the TDV Server > SQL Engine > Optimizations > Data Ship Query > Execution Mode configuration parameter.
DATA_SHIP_MODE

| DATA_SHIP_MODE Syntax | Example |
|---|---|
| DATA_SHIP_MODE="DISABLED" | SELECT {OPTION DATA_SHIP_MODE="DISABLED"} foo FROM... |
| DATA_SHIP_MODE="EXECUTE_FULL_SHIP_ONLY" | SELECT {OPTION DATA_SHIP_MODE="EXECUTE_FULL_SHIP_ONLY"} foo FROM ... |
| DATA_SHIP_MODE="EXECUTE_ORIGINAL" | SELECT {OPTION DATA_SHIP_MODE="EXECUTE_ORIGINAL"} foo FROM... |
| DATA_SHIP_MODE="EXECUTE_PARTIAL_SHIP" | SELECT {OPTION DATA_SHIP_MODE="EXECUTE_PARTIAL_SHIP"} foo FROM ... |

GROUP BY Options

The following query engine hints are available for GROUP BY:

| Option Syntax | Description | Example |
|-----------------|--|--|
| DISABLE_PUSH | DISABLE_PUSH causes the query engine to process the GROUP BY operator locally in TDV Server, instead of pushing it to the data source. If DISABLE_PUSH is not specified, the GROUP BY operator is pushed to the data source whenever possible. | <pre>SELECT MAX(column2) FROM table1 GROUP BY {OPTION DISABLE_PUSH} column1</pre> |
| DISABLE_THREADS | <p>DISABLE_THREADS prevents the query engine from using background threads to speed up processing of the GROUP BY operator. You can use this option to prevent queries from using excessive server resources.</p> <p>If DISABLE_THREADS is not specified, the query engine always uses background threads to speed up processing.</p> <p>This GROUP BY option takes precedence over the SELECT-level DISABLE_THREADS option.</p> | <pre>SELECT MAX(column2) FROM table1 GROUP BY {OPTION DISABLE_THREAD S} column1</pre> |

| Option Syntax | Description | Example |
|-----------------|--|---|
| FORCE_DISK | <p>FORCE_DISK causes the query engine to use disk instead of memory for temporary storage of data that is required to process the GROUP BY operator. This frees up memory for other server operations. FORCE_DISK is particularly useful for queries that consume a large amount of memory.</p> <p>If FORCE_DISK is not specified, the query engine uses memory instead of disk, whenever possible, for maximum performance.</p> <p>This GROUP BY option takes precedence over the SELECT-level option of the same name.</p> | <pre>SELECT MAX(column2) FROM table1 GROUP BY {OPTION FORCE_DISK} column1</pre> |

INSERT, UPDATE, and DELETE Options

The following query engine hints are available for INSERT, UPDATE and DELETE. These options are specified right after the INSERT, UPDATE and DELETE keywords.

| INSERT, UPDATE, DELETE Option | Description | Syntax | Example |
|-------------------------------|---|-------------------------------------|---|
| CASE_SENSITIVE | <p>CASE_SENSITIVE forces string comparisons to be case-sensitive. This option overrides the TDV Server’s Case Sensitivity configuration setting (under TDV Server > SQL Engine > SQL Language).</p> <p>If CASE_SENSITIVE is set to FALSE or not specified, TDV Server’s Case Sensitivity configuration setting determines how string comparisons are evaluated.</p> | CASE_SENSITIVE[={"TRUE" "FALSE"}] | <pre>UPDATE {OPTION CASE_SENSITIVE=" TRUE"} table1 SET column1 = 'BAR' WHERE column1 = 'FOO'</pre> |

| INSERT, UPDATE, DELETE Option | Description | Syntax | Example |
|-------------------------------|---|---|--|
| CHECK_VIEW_CONSTRAINTS | <p>CHECK_VIEW_CONSTRAINTS makes TDV Server preserve the data integrity of the view definition; in other words, it prevents changes to the view.</p> <p>If CHECK_VIEW_CONSTRAINTS is not specified, TDV Server does not preserve the data integrity of the view definition.</p> <p>Suppose a view V1 is defined as follows: SELECT column1 FROM table1 WHERE column1 = 5</p> <p>Suppose also that someone then tries to update V1 with the following update statement: UPDATE V1 SET column1 = 5 WHERE column1 = 6</p> <p>The UPDATE statement fails if CHECK_VIEW_CONSTRAINTS was specified, because a row with value column1=6 is outside the bounds of the definition of the view V1.</p> | CHECK_VIEW_CONSTRAINTS | <pre>UPDATE {OPTION CHECK_VIEW_CONSTRAINTS} table1 SET column1 = 'BAR ' WHERE column1 = 'FOO '</pre> |
| IGNORE_TRAILING_SPACES | <p>IGNORE_TRAILING_SPACES causes comparisons to ignore trailing spaces. This option overrides the TDV Server's Ignore Trailing Spaces configuration setting (under TDV Server > SQL Engine > SQL Language).</p> <p>If IGNORE_TRAILING_SPACES is set to FALSE or not specified, TDV Server's Ignore Trailing Spaces configuration setting determines how string comparisons are evaluated.</p> | IGNORE_TRAILING_SPACES[={"TRUE" "FALSE"}] | <pre>UPDATE {OPTION IGNORE_TRAILING_SPACES="FALSE"} table1 SET column1 = 'BAR ' WHERE column1 = 'FOO '</pre> |

| INSERT, UPDATE, DELETE Option | Description | Syntax | Example |
|-------------------------------------|--|--------|---|
| STRICT | STRICT prevents the query engine from pushing aspects of SQL (such as mathematical and string functions, and the Oracle POSITION function) to the underlying data source when the source does not adhere to strict SQL 92 behavior. This could affect performance. If STRICT is not specified, the query engine relaxes SQL 92 rules to achieve more push. | strict | <pre>UPDATE {OPTION STRICT} table1 SET column2 = 'S' WHERE SIN(column1) = 1</pre> |

JOIN Options

The following query engine hints are available for JOIN.

These options are specified using SQL 92 JOIN syntax. You can also have TDV automatically add them to the query by double-clicking any JOIN line in the execution plan model and making a selection.

- [DISABLE_PUSH \(JOIN Option\), page 252](#)
- [DISABLE_THREADS \(JOIN Option\), page 252](#)
- [FORCE_DISK \(JOIN Option\), page 253](#)
- [FORCE_ORDER \(JOIN Option\), page 253](#)
- [HASH \(JOIN Option\), page 254](#)
- [LEFT_CARDINALITY \(JOIN Option\), page 254](#)
- [NESTEDLOOP \(JOIN Option\), page 255](#)
- [PARTITION_SIZE \(JOIN Option\), page 255](#)
- [RIGHT_CARDINALITY \(JOIN Option\), page 256](#)
- [SEMIJOIN \(JOIN Option\), page 256](#)
- [SORTMERGE \(JOIN Option\), page 257](#)
- [SWAP_ORDER \(JOIN Option\), page 257](#)

DISABLE_PUSH (JOIN Option)

DISABLE_PUSH causes the query engine to process the JOIN operator locally instead of pushing it to the data source. If DISABLE_PUSH is not specified, the JOIN operator is pushed to the data source whenever possible.

Operator

JOIN

Syntax

disable_push

Example

```
SELECT column1 FROM table1 INNER {OPTION DISABLE_PUSH}  
JOIN table2 ON table1.id = table2.id
```

DISABLE_THREADS (JOIN Option)

DISABLE_THREADS can be used to prevent the query engine from using background threads to speed up processing of queries. You can use this option to prevent resource-intensive queries from using excessive server resources.

If DISABLE_THREADS is not specified, the query engine always uses background threads to speed up processing.

This JOIN option takes precedence over the SELECT-level DISABLE_THREADS option.

Operator

JOIN

Syntax

disable_threads

Example

```
SELECT column1 FROM table1 INNER {OPTION DISABLE_THREADS}  
JOIN table2 ON table1.id = table2.id SELECT column1 FROM table2
```


FORCE_DISK (JOIN Option)

FORCE_DISK causes the query engine to use disk rather than memory for temporary storage of the data required to process the JOIN operator. This frees up memory for other server operations. It is useful for queries that consume a large amount of memory and affect performance of other processes running on the server.

If FORCE_DISK is not specified, the query engine uses memory rather than disk, whenever possible, to maximize performance.

This option takes precedence over the SELECT-level FORCE_DISK option.

Operator

JOIN

Syntax

force_disk

Example

```
SELECT column1 FROM table1 INNER {OPTION FORCE_DISK} JOIN table2
ON table1.id = table2.id
```

FORCE_ORDER (JOIN Option)

FORCE_ORDER causes the query optimizer to honor the order of the joins specified in the SQL statement, rather than reordering the join. If FORCE_ORDER is not specified, the optimizer might switch the order of joins to improve the query execution plan.

This is currently used to prevent:

- Union join flipping
- Join reordering
- Reordering of join while selecting the join algorithm, even if a cardinality estimate is provided.

For information on SQL join reordering, see the *TDV User Guide*.

Operator

JOIN

Syntax

force_order

Example

```
SELECT column1 FROM table1 INNER {OPTION FORCE_ORDER}  
JOIN table2 ON table1.id = table2.id
```

HASH (JOIN Option)

HASH causes the optimizer to choose a hash algorithm, if possible, for the join. If HASH is not specified, the optimizer chooses the best algorithm for the join.

Operator

JOIN

Syntax

hash

Example

```
SELECT column1 FROM table1 INNER {OPTION HASH} JOIN table2  
ON table1.id = table2.id
```

LEFT_CARDINALITY (JOIN Option)

LEFT_CARDINALITY provides a cardinality hint for the left-hand side (LHS) of a join. The optimizer uses this option's value as a hint to help choose a better query execution plan.

If LEFT_CARDINALITY is not specified, the optimizer relies on statistics processing for cardinality estimates.

Operator

JOIN

Syntax

LEFT_CARDINALITY=<int>

The <int> argument specifies the cardinality value to use for the left-hand side.

Example

```
SELECT column1 FROM table1 INNER {OPTION LEFT_CARDINALITY=10}  
JOIN table2 ON table1.id = table2.id
```

NESTEDLOOP (JOIN Option)

NESTEDLOOP forces the optimizer to choose a nested-loop algorithm for the join. If you do not specify NESTEDLOOP, the optimizer chooses the best algorithm for the join.

Operator

JOIN

Syntax

nestedloop

Example

```
SELECT column1 FROM table1 INNER {OPTION NESTEDLOOP}
JOIN table2 ON table1.id = table2.id
```

PARTITION_SIZE (JOIN Option)

PARTITION_SIZE restricts the size of the condition clause submitted to the right-hand side (RHS) of a semijoin by specifying the maximum number of condition arguments that can be sent in each batch request. This can be advantageous if a large cardinality result set is expected from the left-hand side (LHS) of a semijoin, and the RHS SQL SELECT statement must be limited in size. This option is also useful in cases where data resources are limited, such as when the SQL string cannot exceed a certain length.

To limit the partition size sent to the RHS, set PARTITION_SIZE to an integer representing the number of arguments in the condition clause submitted to the second data source.

Note: Limiting the number of arguments permitted in the condition clause does not guarantee an acceptably short SQL string, but it does provide adequate control of the submission to avoid problems.

Operator

JOIN

Syntax

PARTITION_SIZE=<int>

The <int> argument specifies the number of arguments in the condition clause submitted to the second data source.

Example

```
SELECT TableX.col1 FROM /Folder/SomeResource/DatabaseX TableX
INNER {OPTION PARTITION_SIZE=9} JOIN
/FolderY/ResourceZ TableY.col2 ON TableX.oid = TableY.oid
```

RIGHT_CARDINALITY (JOIN Option)

RIGHT_CARDINALITY provides a cardinality hint for the right-hand side (RHS) of a join. The optimizer uses this option's value as a hint to help choose a better query execution plan.

If RIGHT_CARDINALITY is not specified, the optimizer relies on statistics processing for cardinality estimates.

Operator

JOIN

Syntax

RIGHT_CARDINALITY=<int>

The <int> argument specifies the cardinality value to use for the right-hand side.

Example

```
SELECT column1 FROM table1 INNER {OPTION RIGHT_CARDINALITY=10000}
JOIN table2 ON table1.id = table2.id
```

SEMIJOIN (JOIN Option)

SEMIJOIN causes the optimizer to try to perform a semijoin optimization. If SEMIJOIN is not specified, the optimizer decides whether to apply semijoin optimization.

Note: Semijoin is an Information Integration tool. It is a fast algorithm that reduces the number of rows retrieved from the right-hand side (RHS). It rewrites the FETCH pushed to the second data source. For this it uses selective criteria provided by the unique values returned from an initial query on the left-hand side (LHS). In a semijoin, LHS is evaluated and loaded into a table in memory, and its cardinality is evaluated. If the cardinality is small enough, an IN clause or an OR expression is created containing all the values in the join criteria from LHS. The clause or expression is then appended to the WHERE clause on RHS and pushed to the database. In this way, only rows with matches are retrieved from RHS.

The semijoin can only be attempted if the RHS can be queried as a single node that fetches against a data source that supports an IN clause or an OR expression.

Operator

JOIN

Syntax

semijoin

Example

```
SELECT column1 FROM table1 INNER {OPTION SEMIJOIN} JOIN table2 ON table1.id = table2.id
```

SORTMERGE (JOIN Option)

SORTMERGE causes the optimizer to consider the sort-merge algorithm when choosing an algorithm for evaluating the join.

If SORTMERGE is set to FALSE, the sort-merge algorithm is excluded from consideration.

Operator

JOIN

Syntax

```
sortmerge[={"TRUE"}|"FALSE"}]
```

Example

```
SELECT column1 FROM table1 INNER {OPTION SORTMERGE}
JOIN table2 ON table1.id = table2.id
```

SWAP_ORDER (JOIN Option)

SWAP_ORDER swaps the order of the join after the SQL is parsed. This can be useful for queries with complex joins, where swapping join order might be easier than trying to move a large amount of text in the SQL. If SWAP_ORDER is not specified, the parsed join order applies.

Operator

JOIN

Syntax

SWAP_ORDER

Example

```
SELECT column1 FROM table1 INNER {OPTION SWAP_ORDER}
```

JOIN table2 ON table1.id = table2.id

ORDER BY Options

The following query engine hints are available for ORDER BY.

- [DISABLE_PUSH \(ORDER BY Option\), page 258](#)
- [DISABLE_THREADS \(ORDER BY Option\), page 258](#)
- [FORCE_DISK \(ORDER BY Option\), page 259](#)

DISABLE_PUSH (ORDER BY Option)

DISABLE_PUSH forces the ORDER BY operator to be processed locally in TDV Server instead of being pushed to the data source. If DISABLE_PUSH is not specified, the ORDER BY operator is pushed to the data source whenever possible.

Operator

ORDER BY

Syntax

disable_push

Example

```
SELECT column1 FROM table1
ORDER BY {OPTION DISABLE_PUSH} column1
```

DISABLE_THREADS (ORDER BY Option)

DISABLE_THREADS prevents the query engine from using background threads to speed up processing of the ORDER BY operator. You can use this option to prevent resource-intensive queries from using excessive server resources.

If DISABLE_THREADS is not specified, the query engine uses background threads to speed processing.

This ORDER BY option takes precedence over the SELECT-level DISABLE_THREADS option.

Operator

ORDER BY

Syntax

```
disable_threads
```

Example

```
SELECT column1 FROM table1
ORDER BY {OPTION DISABLE_THREADS} column1
```

FORCE_DISK (ORDER BY Option)

FORCE_DISK causes the query engine to use disk instead of memory for temporary storage of the data required to process the ORDER BY operator. This frees up memory for other server operations. FORCE_DISK is useful for queries that consume a large amount of memory and affect performance of other processes running on the server.

If FORCE_DISK is not specified, the query engine uses memory instead of disk, whenever possible, to speed performance.

This ORDER BY option takes precedence over the SELECT-level FORCE_DISK option.

Operator

```
ORDER BY
```

Syntax

```
force_disk
```

Example

```
SELECT column1 FROM table1
ORDER BY {OPTION FORCE_DISK} column1
```

SELECT Options

The following query engine hints are available for SELECT. These options are specified immediately following the SELECT keyword.

Examples

```
SELECT {OPTION FORCE_DISK}
SELECT {OPTION FORCE_DISK="FALSE"}
SELECT {OPTION STRICT}
```

Operator-level options (such as JOIN-level options) override SELECT-level options.

SELECT options should be specified at the root-level of the query. When SELECT options are specified as part of a subquery or subselect, they might not affect the root-level query execution plan.

- [CASE_SENSITIVE \(SELECT Option\), page 260](#)
- [DISABLE_CBO \(SELECT Option\), page 261](#)
- [DISABLE_DATA_CACHE \(SELECT Option\), page 261](#)
- [DISABLE_JOIN_PRUNER \(SELECT Option\), page 262](#)
- [DISABLE_PLAN_CACHE \(SELECT Option\), page 263](#)
- [DISABLE_PUSH \(SELECT Option\), page 263](#)
- [DISABLE_SELECTION_REWRITER \(SELECT Option\), page 263](#)
- [DISABLE_STATISTICS \(SELECT Option\), page 264](#)
- [DISABLE_THREADS \(SELECT Option\), page 264](#)
- [FORCE_DISK \(SELECT Option\), page 265](#)
- [IGNORE_TRAILING_SPACES \(SELECT Option\), page 265](#)
- [MAX_ROWS_LIMIT \(SELECT Option\), page 266](#)
- [ROWS_OFFSET \(SELECT Option\), page 268](#)
- [STRICT \(SELECT Option\), page 269](#)
- [PUSH_NULL_SELECTS \(SELECT OPTION\), page 270](#)
- [DISABLE_CONSTANT_FUNCTION_INLINING \(SELECT OPTION\), page 270](#)
- [DISABLE_UNION_PREAGGREGATOR \(SELECT OPTION\), page 270](#)
- [USE_COMPARABLE_ESTIMATES \(SELECT OPTION\), page 271](#)

CASE_SENSITIVE (SELECT Option)

CASE_SENSITIVE forces string comparisons to be case-sensitive. This option overrides the TDV Server's Case Sensitivity configuration setting (under TDV Server > SQL Engine > SQL Language).

If CASE_SENSITIVE is set to FALSE or not specified, TDV Server's Case Sensitivity configuration setting determines how string comparisons are evaluated.

Note: When SELECT options are specified as part of a subquery or subselect, they might not affect the root-level query execution plan.

Operator

SELECT

Syntax

```
case_sensitive[={ "TRUE"|"FALSE" }]
```

Example

```
SELECT {OPTION CASE_SENSITIVE="TRUE"} *
FROM table1
WHERE column1 = 'FOO'
```

DISABLE_CBO (SELECT Option)

Disabling cost-based optimizations (CBO) forces the execution plan to be generated from rule-based heuristics. DISABLE_CBO causes the query optimizer to ignore any table boundary statistics or other table statistics that might have been gathered; the query optimizer applies only heuristics-based optimizations to the execution plan.

If DISABLE_CBO is not specified, the query optimizer applies cost-based optimizations in addition to heuristics-based optimizations.

Note: When SELECT options are specified as part of a subquery or subselect, they might not affect the root-level query execution plan.

Operator

SELECT

Syntax

```
disable_cbo
```

Example

```
SELECT {OPTION DISABLE_CBO} * FROM table1 INNER JOIN table2 ON table1.id = table2.id
```

DISABLE_DATA_CACHE (SELECT Option)

DISABLE_DATA_CACHE causes the query to ignore cached views. This option is useful for queries that require the latest data rather than cached data.

If this option is not specified, cached data is used whenever available.

Note: When SELECT options are specified as part of a subquery or subselect, they might not affect the root-level query execution plan.

Operator

SELECT

Syntax

disable_data_cache

Example

```
SELECT {OPTION DISABLE_DATA_CACHE} * FROM cachedView1
```

DISABLE_JOIN_PRUNER (SELECT Option)

DISABLE_JOIN_PRUNER

Note: When SELECT options are specified as part of a subquery or subselect, they might not affect the root-level query execution plan.

Operator

SELECT

Syntax

disable_join_pruner

Example

```
SELECT { option DISABLE_JOIN_PRUNER="false" }
  t1.*
from /shared/"myquery"/testdb/my_product t1 inner join
    /shared/"myquery"/testdb/products t2
  on t2.productid = t1.productid
```

Relationship: my_product.productid is the foreign key for products.productid primary key.

Result:

The PK table will participate in pruning. The resolved SQL is:

```
SELECT "t1"."categoryid","t1"."categoryname","t1"."productid","t1"."supplierid"
FROM "tutorial"."my_product" "t1"
```

DISABLE_PLAN_CACHE (SELECT Option)

DISABLE_PLAN_CACHE causes the query engine to prepare a fresh query plan each time it executes the query. If DISABLE_PLAN_CACHE is not specified, the query engine uses a cached plan whenever one is available.

Note: When SELECT options are specified as part of a subquery or subselect, they might not affect the root-level query execution plan.

Operator

SELECT

Syntax

disable_plan_cache

Example

```
SELECT {OPTION DISABLE_PLAN_CACHE} * FROM table1
```

DISABLE_PUSH (SELECT Option)

DISABLE_PUSH causes the SELECT to be processed locally in TDV Server instead of being processed at the data source. If DISABLE_PUSH is not specified, the SELECT is pushed to the data source whenever possible.

Note: When SELECT options are specified as part of a subquery or subselect, they might not affect the root-level query execution plan.

Operator

SELECT

Syntax

disable_push

Example

```
SELECT {OPTION DISABLE_PUSH} column1 FROM table1 INNER JOIN table2 ON table1.id = table2.id
```

DISABLE_SELECTION_REWRITER (SELECT Option)

DISABLE_SELECTION_REWRITER causes the SELECT to remove query hint corruption from unexpected CROSS JOINS by restoring a prior query plan.

Operator

SELECT

Syntax

disable_selection_rewriter

ExampleSELECT {OPTION **DISABLE_SELECTION_REWRITER**}**DISABLE_STATISTICS (SELECT Option)**

DISABLE_STATISTICS causes the query engine to ignore table statistics when preparing a query execution plan. This option can be useful for checking whether statistics gathering improves the query execution plan.

If this option is not specified, the query engine uses all available statistics to optimize the query execution plan.

Note: When SELECT options are specified as part of a subquery or subselect, they might not affect the root-level query execution plan.

Operator

SELECT

Syntax

disable_statistics

Example

```
SELECT {OPTION DISABLE_STATISTICS} * FROM table1
WHERE column1 = 5
```

DISABLE_THREADS (SELECT Option)

DISABLE_THREADS prevents the query engine from using background threads to speed up processing. This option can be used to prevent resource-intensive queries from using excessive TDV resources. If DISABLE_THREADS is not specified, the query engine always uses background threads to speed up processing.

Note: When SELECT options are specified as part of a subquery or subselect, they might not affect the root-level query execution plan.

Operator

SELECT

Syntax

```
disable_threads
```

Example

```
SELECT {OPTION DISABLE_THREADS} *
FROM table1 INNER JOIN table2 ON table1.id = table2.id
INNER JOIN table3 ON table1.id = table3.id
```

FORCE_DISK (SELECT Option)

FORCE_DISK forces the query engine to use disk instead of memory for temporary storage of query data. This frees up memory for other server operations. This option is useful for queries that can consume large amounts of memory and affect performance of other processes running on the server.

If FORCE_DISK is not specified, the query engine uses memory rather than disk whenever possible to maximize performance.

Note: When SELECT options are specified as part of a subquery or subselect, they might not affect the root-level query execution plan.

Operator

```
SELECT
```

Syntax

```
force_disk
```

Example

```
SELECT {OPTION FORCE_DISK} *
FROM table1 INNER JOIN table2 ON table1.id = table2.id
INNER JOIN table3 ON table1.id = table3.id
```

IGNORE_TRAILING_SPACES (SELECT Option)

IGNORE_TRAILING_SPACES causes comparisons to ignore trailing spaces. This option overrides the TDV Server's Ignore Trailing Spaces configuration setting (under TDV Server > SQL Engine > SQL Language).

If IGNORE_TRAILING_SPACES is set to FALSE or not specified, TDV Server's Ignore Trailing Spaces configuration setting determines how string comparisons are evaluated.

Note: When SELECT options are specified as part of a subquery or subselect, they might not affect the root-level query execution plan.

Operator

SELECT

Syntax

ignore_trailing_spaces[={"TRUE"|"false"}]

Example

```
SELECT {OPTION IGNORE_TRAILING_SPACES="FALSE"} *
FROM table1
WHERE column1 = 'FOO '
```

MAX_ROWS_LIMIT (SELECT Option)

MAX_ROWS_LIMIT limits the number of rows returned by a query. This is useful if a user is interested in only the first n rows of the results returned.

This option is often used in conjunction with the ROWS_OFFSET (see [ROWS_OFFSET \(SELECT Option\), page 268](#)). How it works in combination with ROWS_OFFSET, OFFSET, FETCH and the maxRows JDBC/ODBC parameter is shown in examples 2 through 9 at the end of this section.

If this option is not specified, all selected rows are returned.

Operator

SELECT

Syntax

MAX_ROWS_LIMIT=<int>

The <int> argument specifies the maximum number of rows the query is to return.

Remarks

- When SELECT options are specified as part of a subquery or subselect, they might not affect the root-level query execution plan.
- For better performance with row filtering, use OFFSET and FETCH rather than MAX_ROWS_LIMIT and ROWS_OFFSET. The reason is that OFFSET and FETCH are SQL-standard options that are pushed to the data source for pass-through queries. MAX_ROWS_LIMIT and ROWS_OFFSET are TDV-only constructs that always perform filtering in TDV (after a much larger number of rows may have been fetched).
- Refer to the SQL 2008 standard for syntax and usage of OFFSET and FETCH.

Example 1

This is a simple example illustrating syntax.

```
SELECT {OPTION MAX_ROWS_LIMIT=100} * FROM table1
```

Example 2

In this example, `maxRows` is too large to have an effect. `MAX_ROWS_LIMIT` allows 25 rows beyond those skipped by `OFFSET`, and `ROWS_OFFSET` removes the first 10 of those.

Query:

```
SELECT {OPTION ROWS_OFFSET=10, MAX_ROWS_LIMIT=25} * FROM " + tableName + "  
OFFSET 50 FETCH NEXT 40 ROWS ONLY
```

Example 3

In this example, `maxRows` is too large to have an effect. `MAX_ROWS_LIMIT` allows 25 rows beyond those skipped by `OFFSET`, and `ROWS_OFFSET` removes the first 10 of those.

Query:

```
SELECT {OPTION ROWS_OFFSET=10, MAX_ROWS_LIMIT=25} * FROM " + tableName + "  
OFFSET 50 FETCH NEXT 12 ROWS ONLY"
```

Example 4

Query:

```
SELECT {OPTION ROWS_OFFSET=10, MAX_ROWS_LIMIT=25} * FROM " + tableName + "  
OFFSET 50 FETCH NEXT 34 ROWS ONLY
```

Example 5

In this example, `maxRows` is too large to have an effect. `MAX_ROWS_LIMIT` allows 25 rows beyond those skipped by `OFFSET`.

Query:

```
SELECT {OPTION MAX_ROWS_LIMIT=25} * FROM " + tableName + "  
OFFSET 50 FETCH NEXT 34 ROWS ONLY
```

Example 6

Query:

```
SELECT {OPTION MAX_ROWS_LIMIT=25} * FROM " + tableName + "  
OFFSET 50 FETCH NEXT 34 ROWS ONLY
```

Example 7

In this example, `maxRows` is too large to have an effect. `MAX_ROWS_LIMIT` allows 25 rows beyond those skipped by `OFFSET`.

Query:

```
SELECT {OPTION MAX_ROWS_LIMIT=25} * FROM " + tableName + "
OFFSET 50 ROWS
```

Example 8

In this example, `maxRows` is too large to have an effect. `ROWS_OFFSET` removes the first 10 rows beyond those skipped by `OFFSET`.

Query:

```
SELECT {OPTION ROWS_OFFSET=10} * FROM " + tableName + "
OFFSET 50 FETCH NEXT 12 ROWS ONLY
```

Example 9

In this example, `ROWS_OFFSET` removes the first 10 rows beyond those skipped by `OFFSET`, and `maxRows` allows 10 of the remaining rows to be returned.

Query:

```
SELECT {OPTION ROWS_OFFSET=10} * FROM " + tableName + "
OFFSET 50 FETCH NEXT 34 ROWS ONLY
```

ROWS_OFFSET (SELECT Option)

`ROWS_OFFSET` causes the query engine to discard the rows before the specified offset integer, which reduces the returned data set.

The collection of rows returned begins with the row specified by the offset integer. For example, if you include the option `ROWS_OFFSET=5`, the returned rows excludes the first 4 and begins with row 5.

Note: For a discussion of how this option, `MAX_ROWS_LIMIT`, `OFFSET`, `FETCH` and the `maxRows` JDBC/ODBC parameter work together, see [“MAX_ROWS_LIMIT \(SELECT Option\)”](#) on page 182.

Operator

```
SELECT
```

Syntax

```
ROWS_OFFSET=<int>
```

The `<int>` argument specifies the number of rows to discard from the returned data set.

Remarks

- You can combine this option with MAX_ROWS_LIMIT to return a restricted set of rows.
- A query should not use the ROWS_OFFSET option with OFFSET/FETCH pagination.
- For better performance with row filtering, use OFFSET and FETCH rather than MAX_ROWS_LIMIT and ROWS_OFFSET. The reason is that OFFSET and FETCH are SQL-standard options that are pushed to the data source for pass-through queries, while MAX_ROWS_LIMIT and ROWS_OFFSET are TDV-only constructs that always perform filtering in TDV (after a much larger number of rows may have been fetched).
- Refer to the SQL 2008 standard for syntax and usage of OFFSET and FETCH.

Example

```
SELECT {OPTION ROWS_OFFSET=10, MAX_ROWS_LIMIT=25} ID, Details
FROM tableZ order by ID
```

STRICT (SELECT Option)

STRICT prevents the query engine from pushing aspects of SQL (such as mathematical and string functions, and the Oracle POSITION function) to the underlying data source when the source does not adhere to strict SQL 92 behavior. This could affect performance. If STRICT is not specified, the query engine relaxes SQL 92 rules to achieve more push.

Note: When SELECT options are specified as part of a subquery or subselect, they might not affect the root-level query execution plan.

Operator

```
SELECT
```

Syntax

```
strict
```

Example

```
SELECT {OPTION STRICT} TAN(column1) FROM table1
```

PUSH_NULL_SELECTS (SELECT OPTION)

PUSH_NULL_SELECTS is an optimization option to push null scans to the target datasource. This may help queries to push null select to the datasource.

Operator

SELECT

Syntax

push_null_selects

Example

```
SELECT {OPTION PUSH_NULL_SELECTS} TAN(column1) FROM table1
```

DISABLE_CONSTANT_FUNCTION_INLINING (SELECT OPTION)

DISABLE_CONSTANT_FUNCTION_INLINING option is used to disable pre-evaluation of CURRENT_TIMESTAMP, CURRENT_DATE, CURRENT_TIME.

Operator

SELECT

Syntax

disable_constant_function_inlining

Example

```
SELECT {OPTION DISABLE_CONSTANT_FUNCTION_INLINING} TAN(column1) FROM table1
```

DISABLE_UNION_PREAGGREGATOR (SELECT OPTION)

DISABLE_UNION_PREAGGREGATOR option disables behavior that may inject GROUP BY below UNION ALL nodes for min, max and count aggregates.

Operator

SELECT

Syntax

```
disable_union_function_inlining
```

Example

```
SELECT {OPTION DISABLE_UNION_PREAGGREGATOR} TAN(column1) FROM table1
```

USE_COMPARABLE_ESTIMATES (SELECT OPTION)

USE_COMPARABLE_ESTIMATES option is used for getting partition points for varchar columns successfully. The distribution in the SelectableEstimate will therefore resolve to StringIndex corresponding to varchar column.

Operator

```
SELECT
```

Syntax

```
use_comparable_estimates
```

Example

```
SELECT {OPTION USE_COMPARABLE_ESTIMATES} TAN(column1) FROM table1
```

UNION, INTERSECT, and EXCEPT Options

The following query engine hints are available for the three set operations UNION, INTERSECT, and EXCEPT:

- [DISABLE_PUSH \(UNION, INTERSECT, and EXCEPT Option\), page 272](#)
- [FORCE_DISK \(UNION, INTERSECT, and EXCEPT Option\), page 272](#)
- [PARALLEL \(UNION, INTERSECT, and EXCEPT Option\), page 273](#)
- [ROUND_ROBIN \(UNION, INTERSECT, and EXCEPT Option\), page 273](#)
- [SORT_MERGE \(UNION, INTERSECT, and EXCEPT Option\), page 274](#)

DISABLE_PUSH (UNION, INTERSECT, and EXCEPT Option)

DISABLE_PUSH causes UNION, INTERSECT, and EXCEPT operators to be processed locally in TDV Server instead of being pushed to the data source. If DISABLE_PUSH is not specified, UNION, INTERSECT, and EXCEPT operators are pushed to the data source whenever possible.

Operators

UNION, INTERSECT, EXCEPT

Syntax

disable_push

Example

```
SELECT column1 FROM table1
UNION ALL {OPTION DISABLE_PUSH}
SELECT column1 FROM table2
```

FORCE_DISK (UNION, INTERSECT, and EXCEPT Option)

FORCE_DISK causes the query engine to use disk instead of memory for temporary storage of the data required to process UNION, INTERSECT, or EXCEPT operators. This frees memory for other server operations. FORCE_DISK is useful for queries that consume a large amount of memory and affect performance of other processes running on the server.

Note: UNION ALL will not force data to disk unless PARALLEL is also specified in the OPTION.

If FORCE_DISK is not specified, the query engine uses memory instead of disk whenever possible.

When the FORCE_DISK option is specified on the SELECT level of a query, it is applied over all nodes and takes precedence even if FORCE_DISK is set to FALSE elsewhere in the query.

Operators

UNION, INTERSECT, EXCEPT

Syntax

force_disk

Example

```
SELECT column1 FROM table1
UNION {OPTION FORCE_DISK}
```

```
SELECT column1 FROM table2
```

PARALLEL (UNION, INTERSECT, and EXCEPT Option)

PARALLEL, when used for a UNION operator, causes the query engine to stream the left-hand side while buffering the right-hand side in memory using a background thread. (The buffer is unbounded, and fails over to disk if necessary.) This can speed up query performance. The trade-off is that the operator becomes memory-intensive. Use this option only if you believe you can load the result set without reaching the managed memory limit.

If you want to minimize memory use while processing both children in parallel, refer to the [ROUND_ROBIN \(UNION, INTERSECT, and EXCEPT Option\)](#), page 273 to see a description of a technique that maintains a small, bounded buffer in memory for each child.

If the PARALLEL option is not specified, the query engine does not load the right-hand side of the UNION while streaming the left-hand side.

Note: The PARALLEL option applies only to UNION—not to INTERSECT or EXCEPT.

Operators

UNION, UNION ALL

Syntax

parallel

Example

```
SELECT column1 FROM table1
UNION ALL {OPTION PARALLEL}
SELECT column1 FROM table2
```

ROUND_ROBIN (UNION, INTERSECT, and EXCEPT Option)

ROUND_ROBIN sets round robin fetch mode, which wraps each child branch of the UNION with a buffered pipe cursor. Each cursor spawns a background thread to prefetch data into its own buffer. When the query is executed, the UNION operator reads from each child pipe cursor in round-robin fashion.

Note: Specifying a fetch mode with SORTMERGE UNION is not usually advisable, because the algorithm reads from both sides.

Operators

UNION, UNION ALL, UNION with DISTINCT, UNION ALL with DISTINCT

Syntax

```
ROUND_ROBIN=[<int>]
```

The <int> argument specifies the maximum number of rows that can be prefetched into each buffer. Optional. The default value is 1000. The maximum value is 2000.

Example

```
SELECT TableX.col2 FROM /local/resource/DB14/TableX
UNION ALL {OPTION ROUND_ROBIN=1500}
SELECT col2 from TableY
```

SORT_MERGE (UNION, INTERSECT, and EXCEPT Option)

SORT_MERGE causes the optimizer to consider sort-merge when choosing an algorithm for evaluating the statement. This can improve efficiency if you want the final result set to be ordered.

The sort-merge algorithm is considered only when the result of the UNION needs to be ordered, such as when you see a SORT node somewhere above the UNION in your query execution plan. If that is not the case, and you still want option **SORT_MERGE** to apply, you can add an ORDER BY clause at the end of the expression that contains the UNION, or at a level above it.

Note that if a SORT node is present, TDV automatically selects the UNION **SORT_MERGE** algorithm (in other words, no user action is needed). If you set **SORT_MERGE** to FALSE, the UNION **SORT_MERGE** algorithm is not used.

Note: An ORDER BY option is required at the end of the expression or at the level above in order for the sort-merge to apply.

Operators

```
UNION, UNION ALL
```

Syntax

```
SORT_MERGE[={"TRUE"}|{"FALSE"}]
```

Example

```
SELECT column1 FROM table1
UNION ALL {OPTION SORT_MERGE="TRUE"}
ORDER BY column1
```

TDV and Business Directory System Tables

This topic describes TDV and Business Directory system tables, which are used to manage TDV software. This topic does not include all system tables—only those exposed in Studio.

The following sections describe the tables and their schemas:

- [Accessing TDV and Business Directory System Tables, page 278](#)

| System Table | |
|---|---------|
| ALL_BD_RESOURCES, page 279 | BD only |
| ALL_CATALOGS, page 280 | |
| ALL_CATEGORIES, page 280 | BD only |
| ALL_CATEGORY_VALUES, page 281 | BD only |
| ALL_CLASSIFICATIONS, page 281 | BD only |
| ALL_COLUMNS, page 282 | |
| ALL_COMMENTS, page 284 | BD only |
| ALL_CUSTOM_PROPERTIES, page 285 | BD only |
| ALL_CUSTOM_PROPERTY_CLASSIFICATIONS, page 286 | BD only |
| ALL_CUSTOM_PROPERTY_GROUPS, page 287 | BD only |
| ALL_CUSTOM_PROPERTY_GROUPS_ASSOCIATIONS, page 287 | BD only |
| ALL_DATASOURCES, page 288 | |
| ALL_DOMAINS, page 289 | |
| ALL_ENDPOINT_MAPPINGS, page 289 | DM only |
| ALL_FOREIGN_KEYS, page 290 | |
| ALL_GROUPS, page 293 | |

| System Table | |
|--|---------|
| ALL_INDEXES, page 293 | |
| ALL_LINEAGE, page 295 | BD only |
| ALL_PARAMETERS, page 296 | |
| ALL_PRINCIPAL_SET_MAPPINGS, page 298 | DM only |
| ALL_PRIVILEGES, page 299 | BD only |
| ALL_PROCEDURES, page 300 | |
| ALL_PUBLISHED_FOLDERS, page 301 | |
| ALL_RELATIONSHIP_COLUMNS, page 302 | |
| ALL_RELATIONSHIPS, page 304 | |
| ALL_RESOURCES, page 307 | |
| ALL_SCHEMAS, page 308 | |
| ALL_TABLES, page 309 | |
| ALL_USERS, page 311 | |
| ALL_WATCHES, page 312 | BD only |
| ALL_WSDL_OPERATIONS, page 312 | |
| DEPLOYMENT_PLAN_DETAIL_LOG, page 314 | DM only |
| DEPLOYMENT_PLAN_LOG, page 315 | DM only |
| DUAL, page 316 | |
| LOG_DISK, page 317 | |
| LOG_EVENTS, page 317 | |
| LOG_IO, page 318 | |
| LOG_MEMORY, page 319 | |
| SYS_CACHES, page 319 | |

| System Table | |
|--|---------|
| SYS_CLUSTER , page 321 | |
| SYS_DATA_OBJECTS , page 322 | |
| SYS_DATASOURCES , page 322 | |
| SYS_DEPLOYMENT_PLANS , page 324 | DM only |
| SYS_PRINCIPAL_SETS , page 325 | DM only |
| SYS_REQUESTS , page 326 | |
| SYS_RESOURCE_SETS , page 328 | DM only |
| SYS_SESSIONS , page 329 | |
| SYS_SITES , page 331 | DM only |
| SYS_STATISTICS , page 331 | |
| SYS_TASKS , page 333 | |
| SYS_TRANSACTIONS , page 335 | |
| SYS_TRANSIENT_COLUMNS , page 336 | MPP |
| SYS_TRANSIENT_SCHEMAS , page 338 | MPP |
| SYS_TRANSIENT_TABLES , page 338 | MPP |
| SYS_TRIGGERS , page 340 | |
| TEMPTABLE_LOG , page 341 | |
| TRANSACTION_LOG , page 342 | |
| USER_PROFILE , page 344 | |

Accessing TDV and Business Directory System Tables

Most system tables are in the Studio resource tree under /Desktop/Composite Data Services/Databases/system/. Tables unique to Business Directory (and some tables visible also on the Studio resource tree) can be accessed from BD under HELP > SYSTEM TABLES. After opening a system table, you can show its contents, which include selected metadata of resources defined for use by client applications.

Note: System tables are *virtual tables*. They map to a physical database table, a view, a structure in server memory, or a combination of these. TIBCO reserves the right to change the system tables at any time.

For system tables, what you see depends on the rights and privileges you have. Studio users are limited to executing SQL SELECT statements on these tables. The rights and privileges to change system tables are locked, to prevent changes that could compromise functionality and performance.

For several tables, you see no rows unless you have the ACCESS_TOOLS right. If you have this right, you see rows for all resources for which you have the READ privilege. Users with both ACCESS_TOOLS and READ_ALL_STATUS rights can see all rows.

To access a current list of system tables

1. Open Studio as the admin user.
2. In the resource tree, expand /Desktop/Composite Data Services/Databases/system/.
3. Select the system table you want to examine.
4. Double-click the table to open it.
5. Use the workspace pane to review details about the system table.

You can use Studio to view system table data. After opening the system table, click Show Contents.

ALL_BD_RESOURCES

This Business Directory system table provides a list of Business Directory resources.

| Column | TDV JDBC Data Type | Nullabl e | Description |
|------------------------------------|--------------------------|--------------|---|
| RESOURCE_ID | INTEGER | | Resource identifier. |
| RESOURCE_NAME | VARCHAR | | Resource name. |
| RESOURCE_TYPE | VARCHAR | | Resource type. |
| PARENT_DATASOU RCE_ID | INTEGER | | Parent data source identifier. |
| PARENT_DATASOU RCE_NAME | VARCHAR | | Parent data source name. |
| SITE_NAME | VARCHAR | | Site name. |
| PARENT_PATH | VARCHAR | | Resource's parent path. |
| GUID | CHAR | | Global unique identifier for the resource. |
| CREATION_TIMEST AMP | BIGINT | | Resource creation time stamp. |
| MODIFICATION_TI MESTAMP_ON_SITE | BIGINT | | Resource modification time stamp on site. |
| MODIFICATION_TI MESTAMP | BIGINT | | Resource most recent modification time stamp. |
| ANNOTATION | VARCHAR | | Resource annotation. |

ALL_CATALOGS

The ALL_CATALOGS system table exposes all published catalogs to which the current user has access. Users can see catalogs for which they have at least one privilege.

| Column | TDV JDBC Data Type | Nullabl e | Description |
|--------------------|--------------------------|--------------|---|
| CATALOG_ID | INTEGER | | Identifier of the catalog. Primary key. |
| CATALOG_NAME | VARCHAR(255) | | Name of the catalog. |
| DATASOURCE_ID | INTEGER | | Identifier of the data source. |
| DATASOURCE_NAME | VARCHAR(255) | | Name of the data source. |
| BD_DATASOURCE_NAME | VARCHAR(255) | | BD name of the data source. |
| GUID | VARCHAR(36) | | Nearly unique 128-bit identifier. |
| ANNOTATION | VARCHAR(36) | Yes | Annotation for the catalog. |
| OWNER_ID | INTEGER | | Identifier of the user who created or owns the catalog. |
| OWNER | VARCHAR(255) | | User name of the user who created or owns the catalog. |
| PARENT_PATH | VARCHAR(255) | | Path to the parent container. |
| BD_PARENT_PATH | VARCHAR(255) | | BD path to the parent container. |

ALL_CATEGORIES

This Business Directory System table provides a list of BD categories.

| Column | TDV JDBC Data Type | Nullable | Description |
|-------------|-----------------------|----------|----------------------|
| CATEGORY_ID | INTEGER | | Category Identifier. |

| Column | TDV JDBC Data Type | Nullable | Description |
|---------------|-----------------------|----------|----------------|
| CATEGORY_NAME | VARCHAR | | Category name. |

ALL_CATEGORY_VALUES

This table provides a list of values for categories.

| Column | TDV JDBC Data Type | Nullable | Description |
|---------------------|-----------------------|----------|----------------------------|
| CATEGORY_VALUE_ID | INTEGER | | Category value Identifier. |
| CATEGORY_VALUE_NAME | VARCHAR | | Category value name. |
| CATEGORY_ID | INTEGER | | Category Identifier. |
| CATEGORY_NAME | VARCHAR | | Category name. |

ALL_CLASSIFICATIONS

This table provides a list of classifications for resources.

| Column | TDV JDBC Data Type | Nullable | Description |
|-------------------|-----------------------|----------|----------------------------|
| RESOURCE_ID | INTEGER | | Resource identifier. |
| RESOURCE_NAME | VARCHAR | | Resource name. |
| RESOURCE_TYPE | VARCHAR | | Resource type. |
| PARENT_PATH | VARCHAR | | Resource's parent path. |
| CATEGORY_VALUE_ID | INTEGER | | Category value Identifier. |

| Column | TDV JDBC Data Type | Nullable | Description |
|---------------------|--------------------------|----------|----------------------|
| CATEGORY_VALUE_NAME | VARCHAR | | Category value name. |
| CATEGORY_NAME | VARCHAR | | Category name. |

ALL_COLUMNS

The ALL_COLUMNS system table exposes all columns in all published tables in all published data sources to which the current user has access.

| Column | TDV JDBC Data Type | Nullable | Description |
|------------------|--------------------------|----------|---|
| COLUMN_ID | INTEGER | | Identifier of the column. Primary key. |
| COLUMN_NAME | VARCHAR(255) | | Name of the column. |
| DATA_TYPE | VARCHAR(255) | | String representation of the data type. |
| ORDINAL_POSITION | INTEGER | | Position of this column in relation to other columns in the same table. |
| JDBC_DATA_TYPE | SMALLINT | | JDBC/ODBC data types. For JDBC data types refer to: http://java.sun.com/j2se/1.4.2/docs/api/java/sql/Types.html For ODBC data types refer to: http://msdn.microsoft.com/en-us/library/bb630290.aspx |

| Column | TDV JDBC Data Type | Nullab le | Description |
|------------------|--------------------------|--------------|--|
| COLUMN_LENGTH | INTEGER | Yes | For CHAR or VARCHAR columns, the max length allowed. For DECIMAL or NUMERIC columns, the total number of digits is the column length value. If it is not one of these four types, the value is NULL. |
| COLUMN_PRECISION | INTEGER | Yes | For a column of DECIMAL or NUMERIC data type, the value is the number of digits. For a column that is not a DECIMAL or NUMERIC data type, the value is NULL. |
| COLUMN_SCALE | INTEGER | Yes | For a column value of DECIMAL or NUMERIC data type, this is the exponent. |
| COLUMN_RADIX | INTEGER | Yes | 10—for all NUMERIC data types. Null—for all non-numeric data types. |
| NULLABLE | SMALLINT | | Indicates whether the column is nullable: 0—NULL is not allowed. 1—NULL is allowed. 2—Unknown whether NULL is allowed or not. |
| IS_NULLABLE | VARCHAR(255) | | Indicates whether the column is nullable: YES—Column is nullable. NO—Column is not nullable. Blank string is returned if it is not known. |
| TABLE_ID | INTEGER | | Identifier of the table. |
| TABLE_NAME | VARCHAR(255) | | Name of the table. |
| SCHEMA_ID | INTEGER | Yes | Identifier of the schema. |
| SCHEMA_NAME | VARCHAR(255) | Yes | Name of the schema. |

| Column | TDV JDBC Data Type | Nullab le | Description |
|--------------------|--------------------------|--------------|---|
| CATALOG_ID | INTEGER | Yes | Identifier of the catalog. |
| CATALOG_NAME | VARCHAR(255) | Yes | Name of the catalog. |
| DATASOURCE_ID | INTEGER | | Identifier of the data source. |
| DATASOURCE_NAME | VARCHAR(255) | | Name of the data source. |
| BD_DATASOURCE_NAME | VARCHAR(255) | | BD name of the data source. |
| ANNOTATION | VARCHAR(2147483647) | Yes | Annotation for the column. |
| OWNER_ID | INTEGER | | Identifier for the user who created or owns the column. |
| OWNER | VARCHAR(255) | | User name of the person who created or owns the column. |
| PARENT_PATH | VARCHAR(1043) | | Path to the parent container. |
| BD_PARENT_PATH | VARCHAR(1043) | | BD path to the parent container. |

ALL_COMMENTS

This table provides a list of comments for resources.

| Column | TDV JDBC Data Type | Nullable | Description |
|---------------|--------------------------|----------|----------------------|
| RESOURCE_ID | INTEGER | | Resource Identifier. |
| RESOURCE_NAME | VARCHAR | | Resource name. |
| RESOURCE_TYPE | VARCHAR | | Resource type. |

| Column | TDV JDBC Data Type | Nullable | Description |
|--------------|--------------------------|----------|---|
| PARENT_PATH | VARCHAR | | Resource's parent path. |
| COMMENT_ID | INTEGER | | Comment Identifier. |
| CREATED | TIMESTAMP | | Comment creation time stamp. |
| LAST_UPDATED | TIMESTAMP | | Comment last modified time stamp. |
| COMMENT | VARCHAR | | Comment text. |
| AUTHOR | VARCHAR | | Author of the comment. |
| AUTHOR_ID | INTEGER | | Author identifier. |
| DOMAIN_NAME | VARCHAR | | Name of domain in which resource resides. |

ALL_CUSTOM_PROPERTIES

This table provides a list of custom properties.

| Column | TDV JDBC Data Type | Nullable | Description |
|-------------------------------|--------------------------|----------|--------------------------------|
| CUSTOM_PROPERTY_ID | INTEGER | | Custom Property Identifier. |
| CUSTOM_PROPERTY_NAME | VARCHAR | | Custom Property name. |
| CUSTOM_PROPERTY_TYPE | VARCHAR | | Custom Property type. |
| CUSTOM_PROPERTY_EXTENDED_TYPE | VARCHAR | | Custom Property Extended type. |

| Column | TDV JDBC Data Type | Nullable | Description |
|-------------------------------|--------------------------|----------|------------------------------------|
| CUSTOM_PROPERTY_GROUP | VARCHAR | | Custom Property group. |
| CUSTOM_PROPERTY_DEFAULT_VALUE | VARCHAR | | Default value for Custom Property. |

ALL_CUSTOM_PROPERTY_CLASSIFICATIONS

This table provides a list of custom property classifications for resources.

| Column | TDV JDBC Data Type | Nullable | Description |
|-------------------|--------------------------|----------|----------------------------|
| RESOURCE_ID | INTEGER | | Resource identifier. |
| RESOURCE_NAME | VARCHAR | | Resource name. |
| RESOURCE_TYPE | VARCHAR | | Resource type. |
| SITE_NAME | VARCHAR | | Site name. |
| PARENT_PATH | VARCHAR | | Resource's parent path. |
| PROPERTY_ID | INTEGER | | Property Identifier. |
| PROPERTY_NAME | VARCHAR | | Property name. |
| PROPERTY_GROUP_ID | INTEGER | | Property group identifier. |
| PROPERTY_GROUP | VARCHAR | | Property group. |
| PROPERTY_TYPE | VARCHAR | | Property type. |
| PROPERTY_VALUE | VARCHAR | | Property value. |

ALL_CUSTOM_PROPERTY_GROUPS

This table provides a list of custom property groups.

| Column | TDV JDBC Data Type | Nullable | Description |
|------------------|-----------------------|----------|-------------------|
| GROUP_ID | INTEGER | | Group identifier. |
| GROUP_NAME | VARCHAR | | Group name. |
| GROUP_ANNOTATION | VARCHAR | | Group annotation. |

ALL_CUSTOM_PROPERTY_GROUPS_ASSOCIATIONS

This table provides a list of custom property group associations.

| Column | TDV JDBC Data Type | Nullable | Description |
|---------------|-----------------------|----------|-------------------------|
| GROUP_ID | INTEGER | | Group identifier. |
| GROUP_NAME | VARCHAR | | Group name. |
| RESOURCE_NAME | VARCHAR | | Resource name. |
| RESOURCE_TYPE | VARCHAR | | Resource type. |
| SITE_NAME | VARCHAR | | Site name. |
| PARENT_PATH | VARCHAR | | Resource's parent path. |

ALL_DATASOURCES

The ALL_DATASOURCES system table exposes all published data sources to which the current user has access. Users can see those data sources for which they have at least one privilege.

| Column | TDV JDBC Data Type | Nullable | Description |
|-------------------------------|---------------------|----------|--|
| DATASOURCE_ID | INTEGER | | Identifier of the data source. Primary key. |
| DATASOURCE_NAME | VARCHAR(255) | | Name of the data source. |
| BD_DATASOURCE_NAME | VARCHAR(255) | | BD name of the data source. |
| DATASOURCE_TYPE | VARCHAR(255) | | Data type of the data source. The number and variety of supported data source types are growing with each release. |
| GUID | VARCHAR(36) | | Nearly unique 128-bit identifier. |
| ANNOTATION | VARCHAR(2147483647) | Yes | Annotation for the data source. |
| OWNER_ID | INTEGER | | Identifier of the user who created or owns the data source. |
| OWNER | VARCHAR(255) | | User name of the person that owns/created the data source. |
| PARENT_PATH | VARCHAR(2147483647) | | Path to the parent container. |
| DATASOURCE_CREATOR_ID | INTEGER | | Identifier of the user who created this data source. Same as USER_ID in ALL_USERS table. |
| DATASOURCE_CREATION_TIMESTAMP | BIGINT | | Timestamp when the data source was created. |

| Column | TDV JDBC Data Type | Nullable | Description |
|-----------------------------------|--------------------|----------|--|
| DATASOURCE_MODIFIER_ID | INTEGER | | Identifier of the user who last modified this data source. Same as USER_ID in ALL_USERS table. |
| DATASOURCE_MODIFICATION_TIMESTAMP | BIGINT | | Timestamp of the last modification of this data source. |

ALL_DOMAINS

The ALL_DOMAINS system table exposes all domains that have been added to the TDV Server. The default domain is composite, which is installed during product installation.

Users can see their own domain and the domain of any group to which they belong. Users with the READ_ALL_USERS right can see all domains.

| Column | TDV JDBC Data Type | Nullable | Description |
|-------------|---------------------|----------|---|
| DOMAIN_ID | INTEGER | | Identifier of the domain. Primary key. |
| DOMAIN_TYPE | VARCHAR(255) | | Domain type. Possible values: composite, dynamic, ldap. |
| DOMAIN_NAME | VARCHAR(255) | | Name of the domain. |
| ANNOTATION | VARCHAR(2147483647) | Yes | Annotation for the domain. |

ALL_ENDPOINT_MAPPINGS

(Deployment Manager) The ALL_ATTRIBUTE_MAPPINGS system table lists all end-point mapping definitions. Users see no rows unless they have the ACCESS_TOOLS right. Users with this right can see all rows.

Note: Unlike most system tables, this table is under /system/deployment in the Studio resource tree.

| Column | TDV JDBC Data Type | Nullable | Description |
|----------------|---------------------|----------|--|
| TARGET_SITE | VARCHAR(2147483647) | | Name of target site. |
| SOURCE_SITE | VARCHAR(2147483647) | | Name of source site. |
| RESOURCE_PATH | VARCHAR(2147483647) | | Resource path. |
| RESOURCE_TYPE | VARCHAR(2147483647) | | Resource type. |
| ENDPOINT_NAME | VARCHAR(2147483647) | | Name of the end point. |
| ENDPOINT_VALUE | VARCHAR(2147483647) | | Value of the end point. |
| IS_ATTRIBUTE | SMALLINT | | Indicates whether the end point is an attribute. |
| RESOURCE_ID | INTEGER | | Identifier of the resource. |

ALL_FOREIGN_KEYS

The ALL_FOREIGN_KEYS system table exposes foreign keys discovered on all published tables in all the data sources for which the current user has access privileges.

Users can see foreign keys on tables for which they have at least one privilege.

| Column | TDV JDBC Data Type | Nullable | Description |
|--------|--------------------|----------|---|
| FK_ID | INTEGER | | Identifier of the foreign key. Primary key. |

| Column | TDV JDBC Data Type | Nullable | Description |
|-----------------------|--------------------------|----------|--|
| FK_NAME | VARCHAR(255)) | | Name of the foreign key. |
| ORDINAL_POSITION | SMALLINT | | Position of the foreign key column in relation to other columns in the same foreign key table. |
| FK_COLUMN_NAME | VARCHAR(255)) | | Name of the foreign key column. |
| FK_TABLE_ID | INTEGER | | Identifier of the table of the foreign key. |
| FK_TABLE_NAME | VARCHAR(255)) | | Name of the table of the foreign key. |
| FK_SCHEMA_ID | INTEGER | Yes | Identifier of the schema of the foreign key. |
| FK_SCHEMA_NAME | VARCHAR(255)) | Yes | Name of the schema of the foreign key. |
| FK_CATALOG_ID | INTEGER | Yes | Identifier of the catalog of the foreign key. |
| FK_CATALOG_NAME | VARCHAR(255)) | Yes | Name of the catalog of the foreign key. |
| FK_DATASOURCE_ID | INTEGER | | Identifier of the data source of the foreign key. |
| FK_DATASOURCE_NAME | VARCHAR(255)) | | Name of the data source of the foreign key. |
| BD_FK_DATASOURCE_NAME | VARCHAR(255)) | | BD name of the data source of the foreign key. |
| PK_NAME | VARCHAR(255)) | | Name of the primary key. |
| PK_COLUMN_NAME | VARCHAR(255)) | | Name of the column in the table with the primary key. |

| Column | TDV JDBC Data Type | Nullable | Description |
|-----------------------|--------------------------|----------|--|
| PK_TABLE_ID | INTEGER | | Identifier of the table of the primary key. |
| PK_TABLE_NAME | VARCHAR(255)) | | Name of the table of the primary key. |
| PK_SCHEMA_ID | INTEGER | Yes | Identifier of the schema of the primary key. |
| PK_SCHEMA_NAME | VARCHAR(255)) | Yes | Name of the schema of the primary key. |
| PK_CATALOG_ID | INTEGER | Yes | Identifier of the catalog of the primary key. |
| PK_CATALOG_NAME | VARCHAR(255)) | Yes | Name of the catalog of the primary key. |
| PK_DATASOURCE_ID | INTEGER | | Identifier of the data source of the primary key. |
| PK_DATASOURCE_NAME | VARCHAR(255)) | | Name of the data source of the primary key. |
| BD_PK_DATASOURCE_NAME | VARCHAR(255)) | | BD name of the data source of the primary key. |
| OWNER_ID | INTEGER | | Identifier for the owner/creator of the foreign key. |
| OWNER | VARCHAR(255)) | | User name of the owner/creator of the foreign key. |
| PARENT_PATH | VARCHAR(1043)) | | Path to the parent container. |
| BD_PARENT_PATH | VARCHAR(255)) | | BD path to the parent container. |

ALL_GROUPS

The ALL_GROUPS system table exposes all the groups that have been added to TDV Server.

Users can see groups in which they are a member. Users with the READ_ALL_USERS right can see all groups.

| Column | TDV JDBC Data Type | Nullable | Description |
|-------------|-----------------------|----------|--|
| GROUP_ID | INTEGER | | Identifier of the group. Primary key. |
| GROUP_NAME | VARCHAR(255) | | Name of the group. |
| DOMAIN_ID | INTEGER | | Unique domain identifier. |
| DOMAIN_NAME | VARCHAR(255) | | Name of the domain. |
| ANNOTATION | VARCHAR(2147483647) | Yes | Group description. |

ALL_INDEXES

The ALL_INDEXES system table exposes all the indexes on all published tables in published data sources to which the current user has access. Users can see indexes on tables for which they have at least one privilege.

| Column | TDV JDBC Data Type | Nullable | Description |
|------------|-----------------------|----------|--|
| INDEX_ID | INTEGER | | Identifier of the index. Primary key. |
| INDEX_NAME | VARCHAR(255) | | Name of the index. |
| INDEX_TYPE | VARCHAR(11) | | Type of the index, whether primary key or other. |

| Column | TDV JDBC Data Type | Nullable | Description |
|--------------------|-----------------------|----------|--|
| COLUMN_NAME | VARCHAR(255) | | Name of the indexed column. |
| ORDINAL_POSITION | SMALLINT | | Position of the indexed column in relation to other columns in the same index. |
| SORT_ORDER | CHAR(1) | | Sort order: A for ascending or D for descending. |
| TABLE_ID | INTEGER | | Identifier of the table. |
| TABLE_NAME | VARCHAR(255) | | Name of the table. |
| SCHEMA_ID | INTEGER | Yes | Identifier of the schema. |
| SCHEMA_NAME | VARCHAR(255) | Yes | Name of the schema. |
| CATALOG_ID | INTEGER | Yes | Identifier of the catalog. |
| CATALOG_NAME | VARCHAR(255) | Yes | Name of the catalog. |
| DATASOURCE_ID | INTEGER | | Identifier of the data source. |
| DATASOURCE_NAME | VARCHAR(255) | | Name of the data source. |
| BD_DATASOURCE_NAME | VARCHAR(255) | | BD name of the data source. |
| IS_UNIQUE | SMALLINT | | Indicates whether the index returns unique values. |
| IS_PRIMARY_KEY | SMALLINT | | Indicates whether the index is a primary index. |
| OWNER_ID | INTEGER | | Identifier for the owner/creator of the index. |
| OWNER | VARCHAR(255) | | User name of the owner/creator of the index. |
| PARENT_PATH | VARCHAR(1043) | | Path to the parent container. |

| Column | TDV JDBC Data Type | Nullable | Description |
|----------------|-----------------------|----------|----------------------------------|
| BD_PARENT_PATH | VARCHAR(255) | | BD path to the parent container. |

ALL_LINEAGE

This Business Directory system table provides information on lineage for resources.

| Column | TDV JDBC Data Type | Nullable | Description |
|-------------------------------|-----------------------|----------|--------------------------------|
| LINEAGE_RESOURCE_ID | INTEGER | | Resource identifier. |
| LINEAGE_RESOURCE_NAME | VARCHAR | | Resource name. |
| LINEAGE_PARENT_PATH | VARCHAR | | Resource's parent path. |
| LINEAGE_SITE_NAME | VARCHAR | | Site name. |
| LINEAGE_DEPENDENCY_PATH | VARCHAR | | Lineage dependency path. |
| LINEAGE_DEPENDENCY_TYPE | VARCHAR | | Lineage dependency type. |
| LINEAGE_DEPENDENCY_SUBTYPE | VARCHAR | | Lineage dependency subtype. |
| LINEAGE_DEPENDENCY_ATTRIBUTES | VARCHAR | | Lineage dependency attributes. |

ALL_PARAMETERS

The ALL_PARAMETERS system table exposes all the parameters that are used in published procedures to which the current user has access. Users can see procedures for which they have at least one privilege.

| Column | TDV JDBC Data Type | Nullable | Description |
|---------------------|--------------------------|----------|--|
| PARAMETER_ID | INTEGER | | Identifier of the parameter. Primary key. |
| PARAMETER_NAME | VARCHAR(255) | | Name of the parameter. |
| DATA_TYPE | VARCHAR(255) | | String representation of the data type. |
| DIRECTION | SMALLINT | | Value indicates the parameter type: 0—Unknown 1—IN 2—INOUT 3—RESULT 4—OUT 5—RETURN |
| ORDINAL_POSITION | INTEGER | Yes | Position of the parameter in relation to other parameters in the same procedure. |
| JDBC_DATA_TYPE | SMALLINT | | JDBC/ODBC data types. For JDBC data types refer to: http://java.sun.com/j2se/1.4.2/docs/api/java/sql/Types.html . |
| PARAMETER_LENGTH | INTEGER | Yes | For a CHAR or VARCHAR parameter, the maximum length allowed; otherwise NULL. |
| PARAMETER_PRECISION | INTEGER | Yes | Value is the number of digits for DECIMAL or NUMERIC data types. If the data type is not DECIMAL or NUMERIC, it is NULL. |

| Column | TDV JDBC Data Type | Nullable | Description |
|--------------------|--------------------------|----------|--|
| PARAMETER_SCALE | INTEGER | Yes | For a DECIMAL or NUMERIC data type, it is the number of digits. If the data type is not DECIMAL or NUMERIC, it is NULL. |
| PARAMETER_RADIX | INTEGER | Yes | Value is 10 for all numeric data types. For non-numeric data types, it is NULL. |
| NULLABLE | SMALLINT | | Indicates whether the column is nullable: 0—NULL is not allowed. 1—NULL is allowed. 2—Unknown whether NULL is allowed or not. |
| IS_NULLABLE | VARCHAR(255) | | Indicates whether the column is nullable: YES—Column is nullable. NO—Column is not nullable. Blank string is returned if it is not known. |
| PROCEDURE_ID | INTEGER | | Identifier of the procedure. |
| PROCEDURE_NAME | VARCHAR(255) | | Name of the procedure. |
| SCHEMA_ID | INTEGER | Yes | Identifier of the schema. |
| SCHEMA_NAME | VARCHAR(255) | Yes | Name of the schema. |
| CATALOG_ID | INTEGER | Yes | Identifier of the catalog. |
| CATALOG_NAME | VARCHAR(255) | Yes | Name of the catalog. |
| DATASOURCE_ID | INTEGER | | Identifier of the data source. |
| DATASOURCE_NAME | VARCHAR(255) | | Name of the data source. |
| BD_DATASOURCE_NAME | VARCHAR(255) | | BD name of the data source. |

| Column | TDV JDBC Data Type | Nullable | Description |
|----------------|--------------------------|----------|---|
| ANNOTATION | VARCHAR(2147483647) | Yes | Annotation for the parameter. |
| OWNER_ID | INTEGER | | Identifier of the person who created or owns the stored procedure in which the parameter is used. |
| OWNER | VARCHAR(255) | | User name of the person who created or owns the procedure in which the parameter is used. |
| PARENT_PATH | VARCHAR(1043) | | Path to the parent container. |
| BD_PARENT_PATH | VARCHAR(255) | | BD path to the parent container. |

ALL_PRINCIPAL_SET_MAPPINGS

The ALL_PRINCIPAL_SET_MAPPINGS system table lists all principal mapping definitions. Users see no rows unless they have the ACCESS_TOOLS right. Users with this right can see all rows.

Note: Unlike most system tables, this table is under /system/deployment in the Studio resource tree.

| Column | TDV JDBC Data Type | Nullabl e | Description |
|------------------|-----------------------|--------------|------------------------|
| TARGET_SITE | VARCHAR(2147483647) | | Name of target site. |
| SOURCE_SITE | VARCHAR(2147483647) | | Name of source site. |
| SOURCE_PRINCIPAL | VARCHAR(2147483647) | | Source site principal. |
| TARGET_PRINCIPAL | VARCHAR(2147483647) | | Target site principal. |

ALL_PRIVILEGES

This table provides a list of resource privileges.

| Column | TDV JDBC Data Type | Nullable | Description |
|---------------|--------------------------|----------|--|
| RESOURCE_ID | INTEGER | | Identifier of the resource. |
| RESOURCE_NAME | VARCHAR | | Name of the resource. |
| COLUMN_ID | INTEGER | | Identifier of the column, -1 if not a column. |
| COLUMN_NAME | VARCHAR | | Name of the column, NULL if not a column. |
| OWNER_ID | INTEGER | | Identifier of the user who created/owns the resource. Same as USER_ID in the ALL_USERS table. |
| OWNER | VARCHAR | | User name of the user who created/owns the resource. Same as USERNAME in the ALL_USERS table. |
| MEMBER_ID | INTEGER | | Identifier of the user who has privilege on the resource. Same as USER_ID in the ALL_USERS table. |
| MEMBER | VARCHAR | | User name of the user who has privileges on the resource. Same as USERNAME in the ALL_USERS table. |
| MEMBER_TYPE | VARCHAR | | The member type; can be either GROUP or USER. |
| PRIVILEGE | INTEGER | | Privilege bitmask value. |

ALL_PROCEEDURES

The ALL_PROCEEDURES system table exposes all published procedures to which the current user has access. Users can see procedures for which they have at least one privilege.

| Column | TDV JDBC Data Type | Nullabl e | Description |
|--------------------|-----------------------|--------------|--|
| PROCEDURE_ID | INTEGER | | Identifier of the procedure. Primary key. |
| PROCEDURE_NAME | VARCHAR(255) | | Name of the procedure. |
| PROCEDURE_TYPE | SMALLINT | | Procedure type. Possible values: 1—A relational data source. 2—A WSDL type of data source. 3—A flat file. 4—The workspace. 5—An LDAP data source. |
| BD_PROCEDURE_TYPE | CHAR | | BD type of the procedure. |
| SCHEMA_ID | INTEGER | Yes | Identifier of the schema. |
| SCHEMA_NAME | VARCHAR(255) | Yes | Name of the schema. |
| CATALOG_ID | INTEGER | Yes | Identifier of the catalog. |
| CATALOG_NAME | VARCHAR(255) | Yes | Name of the catalog. |
| DATASOURCE_ID | INTEGER | | Identifier of the data source. |
| DATASOURCE_NAME | VARCHAR(255) | | Name of the data source. |
| BD_DATASOURCE_NAME | VARCHAR(255) | | BD name of the data source. |
| SITE_NAME | VARCHAR | | Name of the site. |
| GUID | VARCHAR(36) | | Nearly unique 128-bit identifier. |
| ANNOTATION | VARCHAR(2147483647) | | Annotation for the procedure. |

| Column | TDV JDBC Data Type | Nullabl e | Description |
|----------------------------------|--------------------|--------------|--|
| OWNER_ID | INTEGER | | Identifier of the person who created or owns the procedure. |
| OWNER | VARCHAR(255) | | User name of the person who created or owns the procedure. |
| PARENT_PATH | VARCHAR(787) | | Path to the parent container. |
| BD_PARENT_PATH | VARCHAR | | BD path to the parent container. |
| PROCEDURE_CREATOR_ID | INTEGER | | Identifier of the user who created this procedure. Same as USER_ID in ALL_USERS. |
| PROCEDURE_CREATION_TIMESTAMP | BIGINT | | Timestamp when the procedure was created. |
| PROCEDURE_MODIFIER_ID | INTEGER | | Identifier of the user who last modified this procedure. Same as USER_ID in ALL_USERS. |
| PROCEDURE_MODIFICATION_TIMESTAMP | BIGINT | | Timestamp when the procedure was modified. |
| LAST_MODIFICATION_TIMESTAMP | BIGINT | | Timestamp when the procedure was last modified. |

ALL_PUBLISHED_FOLDERS

The ALL_PUBLISHED_FOLDERS system table exposes all of the user-created folders under /services.

| Column | TDV JDBC Data Type | Nullabl e | Description |
|-------------|---------------------|--------------|--------------------------------|
| FOLDER_ID | INTEGER | | ID of the folder. Primary key. |
| FOLDER_NAME | VARCHAR(2147483647) | | Name of the folder. |

| Column | TDV JDBC Data Type | Nullable | Description |
|----------------|---------------------|----------|---|
| GUID | CHAR(2147483647) | | Nearly unique 128-bit identifier. |
| ANNOTATION | VARCHAR(2147483647) | | Annotation for the folder. |
| OWNER_ID | INTEGER | | ID of the person who created/owns the folder. Same as USER_ID in ALL_USERS. |
| OWNER | VARCHAR(255) | | Name of the person who created/owns the folder. Same as USER_NAME in ALL_USERS. |
| PARENT_PATH | VARCHAR(2147483647) | | Path to the parent container. |
| BD_PARENT_PATH | VARCHAR(255) | | BD path to the parent container. |

ALL_RELATIONSHIP_COLUMNS

The ALL_RELATIONSHIP_COLUMNS system table exposes the columns of all relationships to which the current user has access. Users can see relationship columns if they have privileges on the tables involved.

For further information about this system table, see the *Discovery User Guide*.

| Column | TDV JDBC Data Type | Nullable | Description |
|-----------------------|--------------------|----------|---|
| RELATIONSHIP_ID | INTEGER | | Identifier of the relationship. |
| ORDINAL_POSITION | INTEGER | | The order in which this column appears in the relationship. |
| FROM_COLUMN_ID | INTEGER | | Identifier of the “from” column in the relationship. |
| FROM_COLUMN_NAME | VARCHAR(255) | | Name of the “from” column in the relationship. |
| FROM_COLUMN_DATA_TYPE | VARCHAR(255) | | Data type of the “from” column in the relationship. |

| Column | TDV JDBC Data Type | Nullable | Description |
|----------------------|--------------------|----------|---|
| FROM_TABLE_ID | INTEGER | | Identifier of the “from” table in the relationship. |
| FROM_TABLE_NAME | VARCHAR(255) | | Name of the “from” table in the relationship. |
| FROM_SCHEMA_ID | INTEGER | Yes | Identifier of the “from” schema in the relationship. |
| FROM_SCHEMA_NAME | VARCHAR(255) | Yes | Name of the “from” schema in the relationship. |
| FROM_CATALOG_ID | INTEGER | Yes | Identifier of the “from” catalog in the relationship. |
| FROM_CATALOG_NAME | VARCHAR(255) | Yes | Name of the “from” catalog in the relationship. |
| FROM_DATASOURCE_ID | INTEGER | | Identifier of the “from” data source in the relationship. |
| FROM_DATASOURCE_NAME | VARCHAR(255) | | Name of the “from” data source in the relationship. |
| TO_COLUMN_ID | INTEGER | | Identifier of the “to” column in the relationship. |
| TO_COLUMN_NAME | VARCHAR(255) | | Name of the “to” column in the relationship. |
| TO_COLUMN_DATA_TYPE | VARCHAR(255) | Yes | Data type of the “to” column in the relationship. |
| TO_TABLE_ID | INTEGER | | Identifier of the “to” table in the relationship. |
| TO_TABLE_NAME | VARCHAR(255) | | Name of the “to” table in the relationship. |
| TO_SCHEMA_ID | INTEGER | Yes | Identifier of the “to” schema in the relationship. |
| TO_SCHEMA_NAME | VARCHAR(255) | Yes | Name of the “to” schema in the relationship. |

| Column | TDV JDBC Data Type | Nullabl e | Description |
|-----------------------|---------------------|--------------|---|
| TO_CATALOG_ID | INTEGER | Yes | Identifier of the “to” catalog in the relationship. |
| TO_CATALOG_NAME | VARCHAR(255) | Yes | Name of the “to” catalog in the relationship. |
| TO_DATASOURCE_ID | INTEGER | | Identifier of the “to” data source in the relationship. |
| TO_DATASOURCE_NAME | VARCHAR(255) | | Name of the “to” data source in the relationship. |
| OWNER_ID | INTEGER | | Identifier of the person who created or owns the procedure. |
| OWNER | VARCHAR(255) | | User name of the person who created or owns the procedure. |
| FROM_DATA_OBJECT_NAME | VARCHAR(2147483647) | | Name of the “from” data object in the relationship. |
| TO_DATA_OBJECT_NAME | VARCHAR(2147483647) | | Name of the “to” data object in the relationship. |

ALL_RELATIONSHIPS

The ALL_RELATIONSHIPS system table exposes all relationships to which the current user has access. Users can see relationships if they have privileges on the tables involved.

For further information about this system table, see the *Discovery User Guide*.

| Column | TDV JDBC Data Type | Nullabl e | Description |
|--------------------------|--------------------|--------------|---------------------------------|
| RELATIONSHIP_ID | INTEGER | | Identifier of the relationship. |
| RELATIONSHIP_TYPE | VARCHAR(40) | | Relationship type. |
| RELATIONSHIP_CARDINALITY | VARCHAR(32) | | Relationship cardinality. |

| Column | TDV JDBC Data Type | Nullab le | Description |
|--------------------------|-----------------------|--------------|---|
| RELATIONSHIP_STAT US | VARCHAR(40) | | Relationship status. |
| FROM_TABLE_ID | INTEGER | | Identifier of the “from” table in the relationship. |
| FROM_TABLE_NAME | VARCHAR(255) | | Name of the “from” table in the relationship. |
| FROM_SCHEMA_ID | INTEGER | Yes | Identifier of the “from” schema in the relationship. |
| FROM_SCHEMA_NA ME | VARCHAR(255) | Yes | Name of the “from” schema in the relationship. |
| FROM_CATALOG_ID | INTEGER | Yes | Identifier of the “from” catalog in the relationship. |
| FROM_CATALOG_NA ME | VARCHAR(255) | Yes | Name of the “from” catalog in the relationship. |
| FROM_DATASOURCE _ID | INTEGER | | Identifier of the “from” data source in the relationship. |
| FROM_DATASOURCE _NAME | VARCHAR(255) | | Name of the “from” data source in the relationship. |
| TO_TABLE_ID | INTEGER | | Identifier of the “to” table in the relationship. |
| TO_TABLE_NAME | VARCHAR(255) | | Name of the “to” table in the relationship. |
| TO_SCHEMA_ID | INTEGER | Yes | Identifier of the “to” schema in the relationship. |
| TO_SCHEMA_NAME | VARCHAR(255) | Yes | Name of the “to” schema in the relationship. |
| TO_CATALOG_ID | INTEGER | Yes | Identifier of the “to” catalog in the relationship. |
| TO_CATALOG_NAME | VARCHAR(255) | Yes | Name of the “to” catalog in the relationship. |

| Column | TDV JDBC Data Type | Nullable | Description |
|-------------------------|--------------------|----------|--|
| TO_DATASOURCE_ID | INTEGER | | Identifier of the “to” data source in the relationship. |
| TO_DATASOURCE_NAME | VARCHAR(255) | | Name of the “to” data source in the relationship. |
| NUM_MATCHES | INTEGER | | Number-of-matches factor used in calculating a relationship probability score. |
| KEY_FACTOR | NUMERIC(7,4) | | Index key factor used in calculating a relationship probability score. |
| NAME_FACTOR | NUMERIC(7,4) | | Column name comparison factor used in calculating a relationship probability score. |
| MATCH_PERCENTAGE_FACTOR | NUMERIC(7,4) | | Match percentage factor used in calculating a relationship probability score. |
| LOCALITY_FACTOR | NUMERIC(7,4) | | Schema locality factor used in calculating a relationship probability score. |
| KEY_FACTOR_WEIGHT | NUMERIC(7,4) | | Percentage importance to apply to KEY_FACTOR when calculating a relationship probability score. |
| NAME_FACTOR_WEIGHT | NUMERIC(7,4) | | Percentage importance to apply to NAME_FACTOR when calculating a relationship probability score. |
| NUM_MATCHES_WEIGHT | NUMERIC(7,4) | | Percentage importance to apply to NUM_MATCHES when calculating a relationship probability score. |
| MATCH_PERCENTAGE_WEIGHT | NUMERIC(7,4) | | Percentage importance to apply to MATCH_PERCENTAGE_FACTOR when calculating a relationship probability score. |

| Column | TDV JDBC Data Type | Nullable | Description |
|-----------------|--------------------|----------|--|
| LOCALITY_WEIGHT | NUMERIC(7,4) | | Percentage importance to apply to LOCALITY_FACTOR when calculating a relationship probability score. |
| SCORE | NUMERIC(7,4) | | Relationship probability score. |
| SCAN_ID | INTEGER | | Identifier for the scan that created the relationship. |
| OWNER_ID | INTEGER | | Identifier for the person who created or owns the procedure. |
| OWNER | VARCHAR(255) | | User name of the person who created or owns the procedure. |
| CID | INTEGER | | For internal use only. |

ALL_RESOURCES

The ALL_RESOURCES system table exposes all TDV resources to which the current user has access.

Users cannot see any rows from this table unless they have the ACCESS_TOOLS right. All resources are shown for administrators with the READ_ALL_RESOURCES right. Users without the READ_ALL_RESOURCES right can view resource rows in the system table for which they have read privileges both on the resource and on all parent nodes of that resource.

For performance reasons, column and parameter metadata are not returned.

| Column | TDV JDBC Data Type | Nullable | Description |
|---------------|--------------------|----------|--|
| RESOURCE_ID | INTEGER | | Identifier of the resource. Primary key. |
| RESOURCE_NAME | VARCHAR(255) | | Name of the resource. |
| RESOURCE_TYPE | VARCHAR(255) | | Type of the resource. |

| Column | TDV JDBC Data Type | Nullable | Description |
|------------------|--------------------|----------|---|
| ANNOTATION | VARCHAR(65535) | Yes | Annotation for the resource. |
| DEFINITION | VARCHAR(16777215) | Yes | Definition of the resource. Applicable only to certain resources such as SQL Scripts, packaged queries, XSLT-based transformations. |
| OWNER_ID | INTEGER | | Identifier of the user who created or owns the data source. |
| OWNER | VARCHAR(60) | | User name of the person that owns/created the data source. |
| PARENT_PATH | VARCHAR(65535) | | Path to the parent container. |
| GUID | VARCHAR(65535) | | Nearly unique 128-bit identifier. |
| RESOURCE_SUBTYPE | VARCHAR(255) | | Subtype of the resource. |

ALL_SCHEMAS

The ALL_SCHEMAS system table exposes all published schemas to which the current user has access. Users can see schemas for which they have at least one privilege.

| Column | TDV JDBC Data Type | Nullable | Description |
|--------------------|--------------------|----------|--|
| SCHEMA_ID | INTEGER | | Identifier of the schema. Primary key. |
| SCHEMA_NAME | VARCHAR(255) | | Name of the schema. |
| CATALOG_ID | INTEGER | Yes | Identifier of the catalog. |
| CATALOG_NAME | VARCHAR(255) | Yes | Name of the catalog. |
| DATASOURCE_ID | INTEGER | | Identifier of the data source. |
| BD_DATASOURCE_NAME | VARCHAR(255) | | BD name of the data source. |

| Column | TDV JDBC Data Type | Nullable | Description |
|-----------------|---------------------|----------|--|
| DATASOURCE_NAME | VARCHAR(255) | | Name of the data source. |
| GUID | VARCHAR(36) | | Nearly unique 128-bit identifier. |
| ANNOTATION | VARCHAR(2147483647) | Yes | Annotation for the schema. |
| OWNER_ID | INTEGER | | Identifier of the user who created or owns the schema. |
| OWNER | VARCHAR(255) | | User name of the user who created or owns the schema. |
| PARENT_PATH | VARCHAR(531) | | Path to the parent container. |
| BD_PARENT_PATH | VARCHAR(531) | | BD path to the parent container. |

ALL_TABLES

The ALL_TABLES system table exposes all published tables to which the current user has access. Users can see tables for which they have at least one privilege.

| Column | TDV JDBC Data Type | Nullable | Description |
|---------------|--------------------|----------|---------------------------------------|
| TABLE_ID | INTEGER | | Identifier of the table. Primary key. |
| TABLE_NAME | VARCHAR(255) | | Name of the table. |
| TABLE_TYPE | VARCHAR(24) | | Data type of the table. |
| BD_TABLE_TYPE | VARCHAR(24) | | BD table type. |
| SCHEMA_ID | INTEGER | Yes | Identifier of the schema. |
| SCHEMA_NAME | VARCHAR(255) | Yes | Name of the schema. |
| CATALOG_ID | INTEGER | Yes | Identifier of the catalog. |
| CATALOG_NAME | VARCHAR(255) | Yes | Name of the catalog. |

| Column | TDV JDBC Data Type | Nullable | Description |
|------------------------------|---------------------|----------|--|
| DATASOURCE_ID | INTEGER | | Identifier of the data source. |
| DATASOURCE_NAME | VARCHAR(255) | | Name of the data source. |
| BD_DATASOURCE_NAME | VARCHAR(255) | | BD name of the data source. |
| GUID | VARCHAR(36) | | Nearly unique 128-bit identifier. (CHAR in BD.) |
| ANNOTATION | VARCHAR(2147483647) | Yes | Annotation for the table. |
| OWNER_ID | INTEGER | | Identifier of the person who created or owns the table. |
| OWNER | VARCHAR(255) | | Name of the person who created or owns the table. |
| PARENT_PATH | VARCHAR(787) | | Path to the parent container. |
| BD_PARENT_PATH | VARCHAR(787) | | BD path to the parent container. |
| TABLE_CREATOR_ID | INTEGER | | Identifier of the user who created this table. Same as USER_ID in ALL_USERS. |
| TABLE_CREATION_TIMESTAMP | BIGINT | | Timestamp when the table was created. |
| TABLE_MODIFIER_ID | INTEGER | | Identifier of the user who last modified this table. Same as USER_ID in ALL_USERS. |
| TABLE_MODIFICATION_TIMESTAMP | BIGINT | | Timestamp when the table was modified. |
| LAST_MODIFICATION_TIMESTAMP | BIGINT | | Timestamp when the table was last modified. |

ALL_USERS

The ALL_USERS system table exposes all the users in all the domains in the TDV Server. Administrators with the READ_ALL_USERS right can see all users. Users with limited rights can read only their own user rows.

| Column | TDV JDBC Data Type | Nullable | Description |
|-------------|---------------------|----------|--------------------------------------|
| USER_ID | INTEGER | | Identifier of the user. Primary key. |
| USERNAME | VARCHAR(255) | | Log-in name of the user. |
| DOMAIN_ID | INTEGER | | Identifier of user's domain. |
| DOMAIN_NAME | VARCHAR(255) | | Name of user's domain. |
| ANNOTATION | VARCHAR(2147483647) | Yes | Annotation for the user. |

ALL_USER_PROFILES

This table provides a list of user profiles.

| Column | TDV JDBC Data Type | Nullable | Description |
|------------|--------------------|----------|--|
| USER_ID | INTEGER | | User Identifier. |
| FIRST_NAME | VARCHAR | | First name of the user. |
| LAST_NAME | VARCHAR | | Last name of the user. |
| EMAIL | VARCHAR | | Email address of the user. Useful for receiving watch notifications. |
| LOGIN_NAME | VARCHAR | | Login name of the user. |

ALL_WATCHES

This table provides a list of Watches for resources.

| Column | TDV JDBC Data Type | Nullable | Description |
|------------------|--------------------------|----------|--|
| RESOURCE_ID | INTEGER | | Resource identifier. |
| RESOURCE_NAME | VARCHAR | | Resource name. |
| RESOURCE_TYPE | VARCHAR | | Resource type. |
| PARENT_PATH | VARCHAR | | Resource's parent path. |
| WATCH_ID | INTEGER | | Comment identifier. |
| CREATED | TIMESTAMP | | Comment creation time stamp. |
| INCLUDE_CHILDREN | BOOLEAN | | Flag to include watching child resources. |
| OWNER | VARCHAR | | Owner of the watch. |
| OWNER_ID | INTEGER | | Owner identifier. |
| DOMAIN_NAME | VARCHAR | | Name of domain name in which resource resides. |

ALL_WSDL_OPERATIONS

The ALL_WSDL_OPERATIONS system table exposes all published WSDL operations (of Web Services and WSDL data sources) to which the current user has access. Users can see WSDL operations for which they have at least one privilege.

| Column | TDV JDBC Data Type | Nullable | Description |
|----------------|--------------------------|----------|---|
| OPERATION_ID | INTEGER | | Identifier of the operation. Primary key. |
| OPERATION_NAME | VARCHAR(255) | | Name of the operation. |

| Column | TDV JDBC Data Type | Nullabl e | Description |
|----------------------------------|--------------------------|--------------|--|
| DATASOURCE_ID | INTEGER | | Primary key that identifies the data source. |
| DATASOURCE_NAME | VARCHAR(255) | | Name of the data source. |
| BD_DATASOURCE_NAME | VARCHAR(255) | | BD name of the data source. |
| GUID | VARCHAR(36) | | Nearly unique 128-bit identifier. |
| ANNOTATION | VARCHAR(2147483647) | Yes | Annotation for the operation. |
| OWNER_ID | INTEGER | | Identifier of the user who created or owns the WSDL operation. |
| OWNER | VARCHAR(255) | | User name of the user who created or owns the WSDL operation. |
| PARENT_PATH | VARCHAR(2147483647) | | Path to the parent container. |
| BD_PARENT_PATH | VARCHAR(2147483647) | | BD path to the parent container. |
| OPERATION_CREATOR_ID | INTEGER | | Identifier of the user who created this operation. Same as USER_ID in ALL_USERS. |
| OPERATION_CREATION_TIMESTAMP | BIGINT | | Timestamp when the operation was created. |
| OPERATION_MODIFIER_ID | INTEGER | | Identifier of the user who last modified this operation. Same as USER_ID in ALL_USERS. |
| OPERATION_MODIFICATION_TIMESTAMP | BIGINT | | Timestamp when the operation was modified. |

| Column | TDV JDBC Data Type | Nullable | Description |
|-----------------------------|--------------------|----------|---|
| LAST_MODIFICATION_TIMESTAMP | BIGINT | | Timestamp when the operation was last modified. |

DEPLOYMENT_PLAN_DETAIL_LOG

This table provides a list of detailed logs for deployment plan executions. Users see no rows unless they have ACCESS_TOOLS right. If they have this right, they see all rows.

Note: Unlike most system tables, this table is under /system/deployment in the Studio resource tree.

| Column | TDV JDBC Data Type | Nullable | Description |
|------------------------|---------------------|----------|---|
| DEPLOYMENT_PLAN_LOG_ID | INTEGER | | Log identifier of the deployment plan. |
| FROM_SITE | VARCHAR(2147483647) | | Source site. |
| TO_SITE | VARCHAR(2147483647) | | Target site. |
| USER_NAME | VARCHAR(2147483647) | | Name of the user who executed the plan. |
| DEPLOYMENT_PLAN_NAME | VARCHAR(255) | | Name given to the deployment plan. |
| OPERATION_ID | INTEGER | | Identifier of the operation. Primary key. |
| OPERATION_TYPE | VARCHAR(2147483647) | | Operation type. |
| OPERATION_STEP | INTEGER | | Operation step. |
| OPERATION_STEP_TYPE | VARCHAR(2147483647) | | Operation step type. |

| Column | TDV JDBC Data Type | Nullable | Description |
|---------------|---------------------|----------|---|
| START_TIME | TIMESTAMP | | Start time. |
| END_TIME | TIMESTAMP | | End time. |
| CAR | BLOB | | The name of the CAR file that contains the moved resources. |
| RESOURCE_INFO | VARCHAR(2147483647) | | The resources removed from the target site. |
| SETTINGS | VARCHAR(2147483647) | | The settings at the target site during the import process. |
| STATUS | VARCHAR(2147483647) | | Status of the deployment plan. |
| MESSAGE | VARCHAR(2147483647) | | Message to accompany the deployment plan. |

DEPLOYMENT_PLAN_LOG

This table provides a list of deployment plan execution logs. For details such as CAR file name and operation steps, see the DEPLOYMENT_PLAN_DETAIL_LOG table.

Users see no rows unless they have ACCESS_TOOLS right. If they have this right, they see all rows.

Note: Unlike most system tables, this table is under /system/deployment in the Studio resource tree.

| Column | TDV JDBC Data Type | Nullable | Description |
|-----------|---------------------|----------|--|
| LOG_ID | INTEGER | | Log identifier of the deployment plan. |
| FROM_SITE | VARCHAR(2147483647) | | Source site. |
| TO_SITE | VARCHAR(2147483647) | | Target site. |

| Column | TDV JDBC Data Type | Nullable | Description |
|----------------------|---------------------|----------|---|
| DEPLOYMENT_PLAN_ID | INTEGER | | Identifier for the deployment plan. |
| DEPLOYMENT_PLAN_NAME | VARCHAR(255) | | Name given to the deployment plan. |
| USER_NAME | VARCHAR(2147483647) | | Name of the user who executed the plan. |
| START_TIME | TIMESTAMP | | Start time. |
| END_TIME | TIMESTAMP | | End time. |
| STATUS | VARCHAR(2147483647) | | Status of the deployment plan. |
| MESSAGE | VARCHAR(2147483647) | | Message to accompany the deployment plan. |

DUAL

The DUAL system table is a special one-column table with one row. It is similar to the table present in all Oracle database installations. It is useful in situations where the SELECT syntax requires a FROM clause but the query does not require a table.

| Column | TDV JDBC Data Type | Nullable | Description |
|--------|--------------------|----------|---------------------------|
| DUMMY | CHAR(1) | | Value is the character X. |

LOG_DISK

The LOG_DISK system table exposes the log of disk space available on the server. Users see no rows unless they have the ACCESS_TOOLS right.

| Column | TDV JDBC Data Type | Nullable | Description |
|----------------|--------------------|----------|---|
| EVENT_TIME | TIMESTAMP | | The time when the data was logged. |
| CONF_DISK_SIZE | BIGINT | | The size of the disk where conf is located. |
| CONF_DISK_USED | BIGINT | | The amount of space used on the disk. |
| TMP_DISK_SIZE | BIGINT | | The size of the disk where tmp is located. |
| TMP_DISK_USED | BIGINT | | The amount of space used on the disk. |
| LOG_DISK_SIZE | BIGINT | | The size of the disk where logs is located. |
| LOG_DISK_USED | BIGINT | | The amount of space used on the disk. |

LOG_EVENTS

The LOG_EVENTS system table exposes views of events produced by the server. Users see no rows unless they have the ACCESS_TOOLS and READ_ALL_STATUS rights.

| Column | TDV JDBC Data Type | Nullable | Description |
|-----------|--------------------|----------|---|
| EVENT_ID | BIGINT | | The unique ID for this event. |
| PARENT_ID | BIGINT | | The ID for the parent of this event. Same as the EVENT_ID if the event has no parent. |
| TYPE_ID | INTEGER | | The ID of the type of event that occurred. |
| TYPE_NAME | VARCHAR(24) | | A string name for the type of event that occurred. For example, START. |
| CATEGORY | VARCHAR(11) | | A string name for the category of event that occurred. For example, REQUEST. |

| Column | TDV JDBC Data Type | Nullabl le | Description |
|-------------|-------------------------|---------------|---|
| EVENT_TIME | TIMESTAMP | | The time when the data was logged. |
| SEVERITY | VARCHAR(24) | | The severity of the event. |
| OWNER_ID | INTEGER | | The ID of the user who generated the event. |
| OWNER | VARCHAR(255) | | The name of the user who generated the event. |
| DESCRIPTION | VARCHAR(4000) | | The short description of the event. |
| DETAIL | VARCHAR(214748 3647) | | The complete details of the event. |

LOG_IO

The LOG_IO system table exposes the log of I/O produced on the server. Users see no rows unless they have the ACCESS_TOOLS right.

| Column | TDV JDBC Data Type | Nullabl e | Description |
|----------------------|-----------------------------|--------------|---|
| EVENT_TIME | TIMESTA MP | | The time when the data was logged. |
| FROM_CLIENTS | BIGINT | | Estimated number of bytes sent by clients to the server. |
| TO_CLIENTS | BIGINT | | Estimated number of bytes sent by the server to clients. |
| FROM_DATASOURC ES | BIGINT | | Estimated number of bytes sent by data sources to the server. |
| TO_DATASOURCES | BIGINT | | Estimated number of bytes sent by the server to data sources. |

LOG_MEMORY

The LOG_MEMORY system table exposes the log of memory available on the server. Users see no rows unless they have the ACCESS_TOOLS right.

| Column | TDV JDBC Data Type | Nullable | Description |
|---------------|--------------------|----------|---|
| EVENT_TIME | TIMESTAMP | | The time when the data was logged. |
| MEMORY_BYTES | BIGINT | | The amount of Java heap memory used. |
| MEMORY_MAX | BIGINT | | The maximum amount of Java heap memory available. |
| MANAGED_BYTES | BIGINT | | The amount of managed memory used. |
| MANAGED_MAX | BIGINT | | The maximum amount of managed memory available. |

SYS_CACHES

The SYS_CACHES system table provides a list of all cached resources and their current status.

Users see no rows unless they have the ACCESS_TOOLS right. If they have this right, they see rows for all resources for which they have the READ privilege. Users with both ACCESS_TOOLS and READ_ALL_STATUS rights can see all rows.

| Column | TDV JDBC Data Type | Nullable | Description |
|---------------|--------------------|----------|--|
| RESOURCE_ID | INTEGER | | The cached resource ID. |
| RESOURCE_NAME | VARCHAR(255) | | The cached resource name. |
| RESOURCE_TYPE | VARCHAR(255) | | The cached resource type. Can be TABLE or PROCEDURE. |
| OWNER_ID | INTEGER | | The cached resource owner's user ID. |
| OWNER | VARCHAR(255) | | The cached resource owner's name. |

| Column | TDV JDBC Data Type | Nullabl e | Description |
|-----------------------|-----------------------|--------------|---|
| PARENT_PATH | VARCHAR(65535) | | The path to the cached resource. |
| STATUS | VARCHAR(20) | | The status of the cache. Value can be: DISABLED—The cache is disabled. NOT LOADED—The cache is enabled, but not loaded. UP—The cache is enabled and loaded. STALE—The cache is enabled and loaded, but the data has expired DOWN—The cache failed its most recent attempt to load CONFIG ERROR—The cache is not configured properly |
| VARIANT | VARCHAR(255) | Yes | NULL for TABLE views. NULL if no PROCEDURE variants are being tracked. For a PROCEDURE, a comma-separated list of parameter values submitted for generation of the cache. |
| LAST_REFRESH_END | TIMESTAMP | Yes | The time the most recent refresh finished. |
| LAST_SUCCESS_END | TIMESTAMP | Yes | The time the most recent successful refresh finished. |
| LAST_FAIL_END | TIMESTAMP | Yes | The time the most recent failed refresh finished. |
| LAST_ACCESS | TIMESTAMP | Yes | The time the cache was most recently read from. |
| LAST_SUCCESS_DURATION | BIGINT | | The number of milliseconds the most recent successful refresh took to complete. |
| LAST_FAIL_DURATION | BIGINT | | The number of milliseconds the most recent failed refresh took to complete. |
| NUM_SUCCESS | INTEGER | | The number of times the cache was successfully refreshed since the server was started. |
| NUM_FAIL | INTEGER | | The number of times the cache failed to refresh since the server was started. |

| Column | TDV JDBC Data Type | Nullabl e | Description |
|-----------------------|-----------------------|--------------|--|
| NUM_ACCESS | INTEGER | | The number of times the cache was accessed for read since the server was started. |
| STORAGE_USED | BIGINT | | The approximate byte size of the cache data. |
| MESSAGE | VARCHAR(65535) | Yes | A failure message if the cache is in an error state. NULL if there is no message. |
| INITAL_TIME | TIMESTAMP | Yes | The time the trigger is configured to first start. NULL if not condition type TIMER. |
| NEXT_TIME | TIMESTAMP | Yes | The time the trigger will next fire. NULL if not condition type TIMER. |
| FREQUENCY | VARCHAR(255) | Yes | Human-readable description of the frequency of the trigger. NULL if not condition type TIMER. |
| CURRENT_REFRESH_START | TIMESTAMP | Yes | The time the current in-progress refresh started. NULL if not currently refreshing. |
| CURRENT_DURATION | BIGINT | Yes | The number of milliseconds the in-progress refresh has been running. NULL if not currently refreshing. |
| CURRENT_STORAGE | BIGINT | Yes | The approximate byte size of the cache data currently being refreshed. NULL if not currently refreshing. |
| CURRENT_CAUSE | VARCHAR(20) | Yes | The reason the cache is refreshing. NULL if not currently refreshing. Can be MANUAL, SCHEDULED, EXPIRED, or ON_DEMAND. |

SYS_CLUSTER

The SYS_CLUSTER system table provides information about cluster status. It contains one row for each server in the cluster. Users see no rows unless they have the ACCESS_TOOLS and READ_ALL_STATUS rights.

Refer to the *TDV Active Cluster Guide* for more information on the SYS_CLUSTER system table.

SYS_DATA_OBJECTS

The SYS_DATA_OBJECTS system table provides a list of data object definitions. Users see no rows unless they have the ACCESS_TOOLS right. Users with this right can see all rows.

| Column | TDV JDBC Data Type | Nullable | Description |
|-----------------------------|---------------------|----------|--|
| DATA_OBJECT_ID | INTEGER | | Data object identifier. |
| DATA_OBJECT_TYPE | INTEGER | | Data object type. |
| DATA_OBJECT_NAME | VARCHAR(255) | | Data object name. |
| DATA_OBJECT_DESC | VARCHAR(255) | | Data object description. |
| DATA_OBJECT_DEFINITION_NAME | VARCHAR(255) | | Data object definition function name. |
| DATA_OBJECT_DEFINITION_1 | VARCHAR(2147483647) | Yes | Discovery data domain patterns and column. |
| DATA_OBJECT_DEFINITION_2 | VARCHAR(2147483647) | Yes | Discovery data domain transformations. See “Using Data Domains” in the <i>Discovery User Guide</i> . |
| ENABLED | SMALLINT | | Data object enabled flag. |

SYS_DATASOURCES

The SYS_DATASOURCES system table provides a list of all data sources and their current status.

Users see no rows unless they have the ACCESS_TOOLS right. If they have this right, they see rows for all resources for which they have READ privilege. Users with both ACCESS_TOOLS and READ_ALL_STATUS rights can see all rows.

| Column | TDV JDBC Data Type | Nullabl e | Description |
|----------------------|--------------------|--------------|---|
| SOURCE_ID | INTEGER | | The data source's resource ID. |
| SOURCE_NAME | VARCHAR(255) | | The data source's resource name. |
| SOURCE_TYPE | VARCHAR(60) | | The data source's data source type—for example, MySQL. |
| SOURCE_CATEGORY | VARCHAR(60) | | The data source category. Value can be RELATIONAL, FILE, or SERVICE. |
| OWNER_ID | INTEGER | | The data source's resource owner ID. |
| OWNER | VARCHAR(255) | | The data source's resource owner name. |
| PARENT_PATH | VARCHAR(65535) | Yes | The path of the data source resource. Can be NULL for system-owned data sources. |
| STATUS | VARCHAR(20) | | Data source current status: DISABLED—Data source disabled. UP—Data source enabled and running. DOWN—Data source down when last tested. NOT_TESTED—Data source not tested; status unknown. |
| NUM_REQUESTS | INTEGER | | The number of requests processed since the server started. |
| ACTIVE_REQUESTS | INTEGER | | The number of requests currently in progress. |
| MAX_CONN | INTEGER | | The maximum size of the data source's connection pool. |
| NUM_CURRENT_CO NN | INTEGER | | The current size of the data source's connection pool. |

| Column | TDV JDBC Data Type | Nullabl e | Description |
|-----------------|--------------------------|--------------|--|
| NUM_IN_USE_CONN | INTEGER | | The number of data source connections currently in use. |
| NUM_LOGINS | INTEGER | | The number of times new connections were opened since the server started. |
| NUM_LOGOUTS | INTEGER | | The number of times connections were closed since the server started. |
| BYTES_TO | BIGINT | | The estimated number of bytes sent to the data source since the server started. |
| BYTES_FROM | BIGINT | | The estimated number of bytes retrieved from the data source since the server started. |
| MESSAGE | VARCHAR(65535) | Yes | A message about the data source. NULL if no message is available. |

SYS_DEPLOYMENT_PLANS

The SYS_DEPLOYMENT_PLANS system table provides a list of deployment plan definitions. Users see no rows unless they have the ACCESS_TOOLS right. Users with this right can see all rows.

Note: Unlike most system tables, this table is under /system/deployment in the Studio resource tree.

| Column | TDV JDBC Data Type | Nullabl e | Description |
|--------------------------|-----------------------|--------------|-------------------------------------|
| DEPLOYMENT_PLA N_ID | INTEGER | | Identifier for the deployment plan. |
| DEPLOYMENT_PLA N_NAME | VARCHAR(255) | | Name of the deployment plan. |
| TARGET_SITE_NAM E | VARCHAR(255) | | Name of the target site. |

| Column | TDV JDBC Data Type | Nullabl e | Description |
|------------------|-------------------------|--------------|---|
| SOURCE_SITE_NAME | VARCHAR(255) | | Name of the source site. |
| DEFINITION | VARCHAR(2147 483647) | | JSON string defining the deployment plan. |
| ANNOTATION | VARCHAR(2147 483647) | | Annotation. |
| STATUS | VARCHAR(2147 483647) | | Impact status. |
| OWNER | VARCHAR(2147 483647) | | Owner of the deployment plan. |
| CREATE_TIME | BIGINT | | Deployment plan creation time. |
| MODIFY_TIME | BIGINT | | Time of last plan modification. |
| MODIFY_USER | VARCHAR(2147 483647) | | Name of last person to modify the plan. |

SYS_PRINCIPAL_SETS

The SYS_PRINCIPAL_SETS system table provides a list of principal set definitions. Users see no rows unless they have the ACCESS_TOOLS right. Users with this right can see all rows.

Note: Unlike most system tables, this table is under /system/deployment in the Studio resource tree.

| Column | TDV JDBC Data Type | Nullabl e | Description |
|--------------------|-------------------------|--------------|------------------------------|
| PRINCIPAL_SET_NAME | VARCHAR(255) | | Name of the resource set. |
| SITE_NAME | VARCHAR(255) | | Name of the site. |
| DEFINITION | VARCHAR(2147 483647) | | Definition of principal set. |

| Column | TDV JDBC Data Type | Nullabl e | Description |
|-------------|-------------------------|--------------|--|
| ANNOTATION | VARCHAR(2147 483647) | | Annotation. |
| STATUS | VARCHAR(2147 483647) | | Impact status. |
| OWNER | VARCHAR(2147 483647) | | Owner of the principal set. |
| CREATE_TIME | BIGINT | | Principal set creation time. |
| MODIFY_TIME | BIGINT | | Time of last modification to the principal set. |
| MODIFY_USER | VARCHAR(2147 483647) | | Name of last person to modify the principal set. |

SYS_REQUESTS

The SYS_REQUESTS system table provides a list of current and recent requests and their current status.

Users see no rows unless they have the ACCESS_TOOLS right. If they have this right, they see rows for all requests they own. Users with both ACCESS_TOOLS and READ_ALL_STATUS rights can see all rows.

| Column | TDV JDBC Data Type | Nullabl e | Description |
|--------------------|-----------------------|--------------|--|
| REQUEST_ID | BIGINT | | The request's ID. |
| PARENT_ID | BIGINT | Yes | The parent request's ID. NULL if there is no parent request. |
| SESSION_ID | BIGINT | | The request's session ID. |
| TRANSACTION_I D | BIGINT | | The request's transaction ID. |
| OWNER_ID | INTEGER | | The request session's user ID. |
| OWNER | VARCHAR(255) | | The request session's user name. |

| Column | TDV JDBC Data Type | Nullabl e | Description |
|-----------------|-----------------------|--------------|--|
| REQUEST_TYPE | VARCHAR(255) | | The request type. For example, SQL or SQL Script. |
| STATUS | VARCHAR(20) | | The request status can be one of the following: STARTED—The request is in the process of starting. This status usually lasts only a short time. WAITING—The request is waiting for enough system resources to start running. RUNNING—The request is currently executing. READY—The request has completed execution and results are available. CLOSING—The request is in the process of closing. This status usually lasts only a short time. SUCCESS—The request was completed successfully. FAILED—The request failed. TERMINATED—The request was terminated. |
| DESCRIPTION | VARCHAR(65535) | | The request's source, or a description of what was called. |
| START_TIME | TIMESTAMP | | The time when the request started. |
| END_TIME | TIMESTAMP | | The time when the request ended. NULL if it is still running. |
| TOTAL_DURATION | BIGINT | | The number of milliseconds the request required to execute. |
| SERVER_DURATION | BIGINT | | The number of milliseconds of server-side time that elapsed during request execution. |
| ROWS_AFFECTED | BIGINT | | The number of rows affected by the request. For SQL <code>SELECT</code> statements, this is the number of rows read. For other requests, this is the number of rows modified. A value of -1 indicates that the number is not known. |
| MAX_MEMORY | BIGINT | Yes | The maximum amount of memory reserved by the request during execution. |
| MAX_DISK | BIGINT | Yes | The maximum amount of disk used by the request during execution. |
| CURRENT_MEMORY | BIGINT | | The current amount of memory reserved by the request. |

| Column | TDV JDBC Data Type | Nullabl e | Description |
|---------------------|--------------------|--------------|--|
| CURRENT_DISK | BIGINT | Yes | The current amount of disk in use by the request. |
| MESSAGE | VARCHAR(65535) | Yes | A message that is usually set on failure to provide additional information. NULL if no message is available. |
| MAX_USED_MEMORY | BIGINT | | The maximum amount of memory used by the request during execution. |
| CURRENT_USED_MEMORY | BIGINT | | The current amount of memory in use by the request. |

SYS_RESOURCE_SETS

The SYS_RESOURCE_SETS system table provides a list of resource set definitions.

Users see no rows unless they have ACCESS_TOOLS right. If they have this right, they see all rows.

Note: Unlike most system tables, this table is under /system/deployment in the Studio resource tree.

| Column | TDV JDBC Data Type | Nullabl e | Description |
|-------------------|---------------------|--------------|--|
| RESOURCE_SET_NAME | VARCHAR(255) | | Name of the resource set. |
| SITE_NAME | VARCHAR(255) | | Name of the site. |
| DEFINITION | LONGVARCHAR | | JSON string defining the resource set. |
| ANNOTATION | VARCHAR(2147483647) | | Annotation. |
| STATUS | VARCHAR(2147483647) | | Impact status of the resource set. |
| OWNER | VARCHAR(2147483647) | | Owner of the resource set. |

| Column | TDV JDBC Data Type | Nullabl e | Description |
|-------------|-----------------------|--------------|---|
| CREATE_TIME | BIGINT | | Resource set creation time. |
| MODIFY_TIME | BIGINT | | Time of last resource set modification. |
| MODIFY_USER | VARCHAR(2147483647) | | Name of last person to modify the resource set. |

SYS_SESSIONS

The SYS_SESSIONS system table provides a list of current and recent sessions and their current status.

Users see no rows unless they have the ACCESS_TOOLS right. If they have this right, they see rows for all sessions they own. Users with both ACCESS_TOOLS and READ_ALL_STATUS rights see all rows.

| Column | TDV JDBC Data Type | Nullabl e | Description |
|--------------|-----------------------|--------------|---|
| SESSION_ID | BIGINT | | Unique session ID. |
| OWNER_ID | INTEGER | | The ID of the user logged into this session. |
| OWNER | VARCHAR(255) | | The name of the user logged into this session. |
| SESSION_TYPE | VARCHAR(20) | | The session type can be one of the following: HTTP—A web services client. INTERNAL—A session started within the server. JDBC—A JDBC client. ODBC—An ODBC client. STUDIO—The Studio tool. |
| SESSION_NAME | VARCHAR(255) | Yes | The name of the session. NULL if not provided by the client. |
| HOST | VARCHAR(255) | Yes | The host the client is connecting from. NULL for INTERNAL sessions. |

| Column | TDV JDBC Data Type | Nullabl e | Description |
|---------------------|--------------------|--------------|--|
| DATASOURCE_ID | INTEGER | Yes | The data service ID the client is connecting on. NULL if no data service is in use. |
| LOGIN_TIME | TIMESTAMP | | The time at which the session started. |
| LOGOUT_TIME | TIMESTAMP | Yes | The time at which the session ended. NULL if the session is still active. |
| STATUS | VARCHAR(20) | | The session status can be one of the following: ACTIVE—The session is still active. CLOSED—The session was closed in an orderly fashion. DISCONNECTED—The session was disconnected. TERMINATED—The session was terminated. TIMED_OUT—The session timed out. |
| IDLE_DURATION | BIGINT | | The number of milliseconds the session has been idle. |
| TIMEOUT_DURATION | BIGINT | | The number of milliseconds after which the session will time out. |
| TOTAL_REQUESTS | INTEGER | | The number of requests created on this session. |
| ACTIVE_REQUESTS | INTEGER | | The number of requests open on this session. |
| TOTAL_TRANSACTIONS | INTEGER | | The number of transactions created on this session. |
| ACTIVE_TRANSACTIONS | INTEGER | | The number of transactions open on this session. |
| BYTES_TO_CLIENT | BIGINT | | The estimated number of bytes sent to the client. |
| BYTES_FROM_CLIENT | BIGINT | | The estimated number of bytes received from the client. |

SYS_SITES

The SYS_SITES system table provides a list of site definitions.

Users see no rows unless they have the ACCESS_TOOLS right. If they have this right, they see all rows.

Note: Unlike most system tables, this table is under /system/deployment in the Studio resource tree.

| Column | TDV JDBC Data Type | Nullabl e | Description |
|-------------|-----------------------|--------------|---|
| SITE_NAME | VARCHAR(255) | | Name of the site. |
| HOST_NAME | VARCHAR(255) | | Name of the site host. |
| PORT | INTEGER | | Host port through which to connect to the site. |
| DOMAIN | VARCHAR(255) | | Domain of the user who can log in to the site host. |
| USER_NAME | VARCHAR(255) | | Name of the user who can log in to the site host. |
| ANNOTATION | VARCHAR(2147483647) | | Notes about the site. |
| STATUS | VARCHAR(2147483647) | | Impact status. |
| MODIFY_TIME | TIMESTAMP | | Time of last plan modification. |
| OFFLINE | BOOLEAN | | Whether the site is offline (0) or online (1). (BD only.) |

SYS_STATISTICS

The SYS_STATISTICS system table provides a list of current and recent sessions and their current status.

Users see no rows unless they have the ACCESS_TOOLS right. If they have this right, they see rows for all resources for which they have READ privilege. Users with both ACCESS_TOOLS and READ_ALL_STATUS rights can see all rows.

| Column | TDV JDBC Data Type | Nullable | Description |
|-----------------------|--------------------|----------|--|
| RESOURCE_ID | INTEGER | | The resource ID. |
| RESOURCE_NAME | VARCHAR(255) | | The resource name. |
| RESOURCE_TYPE | VARCHAR(255) | | The resource type. Can be TABLE or DATASOURCE. |
| OWNER_ID | INTEGER | | Owner's user ID. |
| OWNER | VARCHAR(255) | | Owner's name. |
| PARENT_PATH | VARCHAR(255) | | Path to the folder that contains the resource. |
| IS_ENABLED | VARCHAR(20) | | Indicates if statistics data will be used. Can be true or false. |
| STATUS | VARCHAR(20) | | Statistics status: STALE, NOT_LOADED, FAILED, UNKNOWN, or UP. |
| LAST_REFRESH_END | TIMESTAMP | | The time the last gather process finished. |
| LAST_SUCCESS_END | TIMESTAMP | | The last time gather process finished successfully. |
| LAST_FAIL_END | TIMESTAMP | | The last time gather process finished with an error. |
| LAST_SUCCESS_DURATION | BIGINT | | Elapsed time (in milliseconds) of the last successful statistics gather process. |
| LAST_FAIL_DURATION | BIGINT | | Elapsed time (in milliseconds) of the last failed statistics gather process. |
| NUM_SUCCESS | INTEGER | | Number of times stats data was successfully refreshed since last server start. |

| Column | TDV JDBC Data Type | Nullabl e | Description |
|---------------------------|--------------------|--------------|---|
| NUM_FAIL | INTEGER | | Number of times statistics data failed to refresh since the last time the server started. |
| MESSAGE | VARCHAR(255) | | Message that provides additional information for some status types. |
| CURRENT_REFRESH_S TART | TIMESTAMP | Yes | The time currently running stats gather process started. NULL if not currently running. |
| CURRENT_DURATION | BIGINT | Yes | Elapsed time of currently running stats gather process. NULL if not currently running. |

SYS_TASKS

The SYS_TASKS system table provides a list of all tasks running in the system. Users see no rows unless they have the ACCESS_TOOLS right. Users with this right can see all rows.

| Column | TDV JDBC Data Type | Nullabl e | Description |
|-------------------|-----------------------|--------------|---|
| TASK_ID | BIGINT | | Task identifier. |
| TASK_CATEGORY | VARCHAR(60) | No | Task category. |
| TASK_TYPE | VARCHAR(255) | No | Task type. |
| NAME | VARCHAR(16777215) | | Task name. |
| RESOURCE_IDS | VARCHAR(16777215) | | Comma-separated list of identifiers of resources involved. |
| FROM_RESOURCE_IDS | VARCHAR(16777215) | | Comma-separated list of identifiers of “from” resources involved. |

| Column | TDV JDBC Data Type | Nullabl e | Description |
|---------------------------|--------------------|--------------|---|
| TO_RESOURCE_IDS | VARCHAR(16777215) | | Comma-separated list of identifiers of “to” resources involved. |
| PARENT_TASK_ID | BIGINT | | Parent task identifier. |
| DEPENDENT_TASK_IDS | VARCHAR(16777215) | | Dependent task identifiers. |
| STATUS | VARCHAR(60) | No | The status of the task. |
| START_TIME | TIMESTAMP | | Time when the task started. |
| END_TIME | TIMESTAMP | | Time when the task ended. |
| DURATION | BIGINT | | Total processing time, in milliseconds. |
| SCAN_ID | INTEGER | | ID for associated groups of tasks. |
| PROCESSING_TIME_REMAINING | BIGINT | | Time remaining to execute this task. |
| TOTAL_TIME_REMAINING | BIGINT | | Time remaining to execute a parent task and all of its offspring. |
| ROWS_PROCESSED | BIGINT | Yes | Number of table rows already processed. |
| OWNER_ID | INTEGER | | ID of the user who created the task. |
| OWNER | VARCHAR(255) | | Name of the user who created the task. |
| ERROR_CODE | INTEGER | Yes | Error code if task failed. |
| ERROR_MESSAGE | VARCHAR(16777215) | Yes | Error message if task failed. |
| FLAGS | INTEGER | | For internal use only. |
| CID | INTEGER | | For internal use only. |

| Column | TDV JDBC Data Type | Nullabl e | Description |
|---------|--------------------|--------------|--|
| CLEARED | BIT | Yes | Blocks display of this task in user interface. |

SYS_TRANSACTIONS

The SYS_TRANSACTIONS system table provides a list of current and recent transactions and their current status.

Users see no rows unless they have the ACCESS_TOOLS right. If they have this right, they see rows for all transactions they own. Users with both ACCESS_TOOLS and READ_ALL_STATUS rights can see all rows.

| Column | TDV JDBC Data Type | Nullabl e | Description |
|----------------|--------------------|--------------|---|
| TRANSACTION_ID | BIGINT | | The unique ID for the transaction to which this log entry applies. |
| SESSION_ID | BIGINT | | The transaction's session ID. |
| OWNER_ID | INTEGER | | The ID of the user logged into this session. |
| OWNER | VARCHAR(255) | | The name of the user logged into this session. |
| MODE | VARCHAR(255) | | The mode of the transaction, which can be: AUTO—The transaction will automatically commit or roll back at the end of the primary request. EXPLICIT—The transaction will not commit or roll back until explicitly told to do so. |
| STATUS | VARCHAR(20) | | Status of the transaction, which can be: ACTIVE—The transaction is still being executed. COMMITTED—The transaction has been committed. ROLLED_BACK—The transaction has been rolled back. TERMINATED—The transaction was terminated. |
| START_TIME | TIMESTAMP | | The time when the transaction was started. |

| Column | TDV JDBC Data Type | Nullable | Description |
|-----------------|--------------------|----------|---|
| END_TIME | TIMESTAMP | Yes | The time when the transaction completed. NULL if it is still in progress. |
| DURATION | BIGINT | | The number of milliseconds the transaction was running. |
| TOTAL_REQUESTS | INTEGER | | The number of requests created in the transaction. |
| ACTIVE_REQUESTS | INTEGER | | The number of requests active in the transaction. |

SYS_TRANSIENT_COLUMNS

Used to hold data for the MPP engine.

| Column | TDV JDBC Data Type | Nullable | Description |
|------------------|--------------------|----------|-------------|
| COLUMN_ID | INTEGER | | |
| COLUMN_NAME | VARCHAR(255) | | |
| DATA_TYPE | VARCHAR(255) | | |
| ORDINAL_POSITION | INTEGER | | |
| JDBC_DATA_TYPE | SMALLINT | | |
| COLUMN_LENGTH | INTEGER | Yes | |
| COLUMN_PRECISION | INTEGER | Yes | |
| COLUMN_SCALE | INTEGER | Yes | |
| COLUMN_RADIX | INTEGER | Yes | |

| Column | TDV JDBC Data Type | Nullable | Description |
|-----------------|--------------------|----------|---|
| NULLABLE | SMALLINT | | Indicates whether the column is nullable -0 if NULL is not allowed -1 if NULL is allowed - 2 if it is unknown |
| IS_NULLABLE | VARCHAR(255) | | Indicates whether the column is nullable - YES if it is nullable -NO if it is not nullable -Blank string is returned if value is not known |
| TABLE_ID | INTEGER | | |
| TABLE_NAME | VARCHAR(255) | | |
| SCHEMA_ID | INTEGER | Yes | |
| SCHEMA_NAME | VARCHAR(255) | Yes | |
| CATALOG_ID | INTEGER | Yes | |
| CATALOG_NAME | VARCHAR(255) | Yes | |
| DATASOURCE_ID | INTEGER | | |
| DATASOURCE_NAME | VARCHAR(255) | | |
| ANNOTATION | VARCHAR(65535) | Yes | Annotation for the column. |
| OWNER_ID | INTEGER | | Identifier for the user who created/owns the column. Same as USER_ID in Table: ALL_USERS |
| CID | INTEGER | | Commit ID |
| HAS_COL_PRIV | SMALLINT | | Not used |

SYS_TRANSIENT_SCHEMAS

Used to hold data for the MPP engine.

| Column | TDV JDBC Data Type | Nullable | Description |
|-----------------|--------------------|----------|--|
| SCHEMA_ID | INTEGER | | Primary key identifier of the schema |
| SCHEMA_NAME | VARCHAR(255) | | |
| CATALOG_ID | INTEGER | Yes | |
| CATALOG_NAME | VARCHAR(255) | Yes | |
| DATASOURCE_ID | INTEGER | | |
| DATASOURCE_NAME | VARCHAR(255) | | |
| ANNOTATION | VARCHAR(65535) | Yes | |
| OWNER_ID | INTEGER | | Identifier for the user who created/owns the column. Same as USER_ID in Table: ALL_USERS |
| CID | INTEGER | | Commit ID |
| GUID | VARCHAR(36) | | 128 bit identifier that is practically unique |

SYS_TRANSIENT_TABLES

Used to hold data for the MPP engine.

| Column | TDV JDBC Data Type | Nullable | Description |
|------------|--------------------|----------|-------------|
| TABLE_ID | INTEGER | | |
| TABLE_NAME | VARCHAR(255) | | |

| Column | TDV JDBC Data Type | Nullable | Description |
|------------------------------|--------------------|----------|---|
| TABLE_TYPE | VARCHAR(255) | | The only possible value of this column is "TABLE". |
| CARDINALITY | INTEGER | Yes | Number of rows in the table since last introspection. If the CARDINALITY is unknown then the value is null. |
| SCHEMA_ID | INTEGER | Yes | |
| SCHEMA_NAME | VARCHAR(255) | Yes | |
| CATALOG_ID | INTEGER | Yes | |
| CATALOG_NAME | VARCHAR(255) | Yes | |
| DATASOURCE_ID | INTEGER | | |
| DATASOURCE_NAME | VARCHAR(255) | | |
| ANNOTATION | | Yes | |
| OWNER_ID | INTEGER | | |
| CID | INTEGER | | Commit ID |
| TABLE_CREATOR_ID | INTEGER | | |
| TABLE_CREATION_TIMESTAMP | BIGINT | | |
| TABLE_MODIFIER_ID | INTEGER | | |
| TABLE_MODIFICATION_TIMESTAMP | BIGINT | | Timestamp of the last modification of this table. |
| GUID | VARCHAR(36) | | 128 bit identifier that is practically unique |

SYS_TRIGGERS

The SYS_TRIGGERS system table provides a list of triggers defined in the system and their current status.

Users see no rows unless they have the ACCESS_TOOLS right. If they have this right, they see rows for all resources they have READ privilege to. Users with both ACCESS_TOOLS and READ_ALL_STATUS rights can see all rows.

| Column | TDV JDBC Data Type | Nullab le | Description |
|--------------------|-----------------------|--------------|--|
| RESOURCE_I D | INTEGER | | The trigger’s resource ID. |
| RESOURCE_N AME | VARCHAR(255) | | The trigger’s resource name. |
| OWNER_ID | INTEGER | | The trigger resource owner ID. |
| OWNER | VARCHAR(255) | | The trigger resource owner name. |
| PARENT_PAT H | VARCHAR(65535) | | The path of the trigger resource. Field length: 65535. |
| PARENT_TYP E | VARCHAR(255) | | The type of the trigger’s parent resource. |
| CONDITION_ TYPE | VARCHAR(60) | | The trigger’s condition type. For example, TIMER. |
| ACTION_TYP E | VARCHAR(60) | | The trigger’s action type. For example, PROCEDURE. |
| STATUS | VARCHAR(20) | | The trigger’s current status: DISABLED—The trigger is disabled. ACTIVE—The trigger is enabled. |
| LAST_TIME | TIMESTAMP | | The most recent time the trigger fired. |
| LAST_SUCCE SS | TIMESTAMP | | The most recent time the trigger succeeded. |
| LAST_FAIL | TIMESTAMP | | The most recent time the trigger failed. |
| NUM_TOTAL | INTEGER | | The number of times the trigger has fired. |

| Column | TDV JDBC Data Type | Nullable | Description |
|-------------|--------------------|----------|--|
| NUM_SUCCES | INTEGER | | The number of times the trigger has succeeded. |
| NUM_FAIL | INTEGER | | The number of times the trigger has failed. |
| INITAL_TIME | TIMESTAMP | Yes | The time the trigger was configured to first start. NULL if not condition type TIMER. |
| NEXT_TIME | TIMESTAMP | Yes | The time the trigger will next fire. NULL if not condition type TIMER. |
| FREQUENCY | VARCHAR(255) | Yes | Human-readable description of the frequency of the trigger. NULL if not condition type TIMER. |
| MESSAGE | VARCHAR(65535) | Yes | A message about the trigger status that is often set on failure. NULL if no message is available. Field length: 65535. |

TEMPTABLE_LOG

The TEMPTABLE_LOG provides a read-only view of all active temporary tables on a specific TDV server node. TDV uses this information during a server restart to clean up any temporary tables left behind when a server is shut down or killed during a transaction.

Users need ACCESS_TOOLS and READ_ALL_STATUS rights to see the table rows.

| Column | TDV JDBC Data Type | Nullable | Description |
|------------------------|---------------------|----------|--|
| SESSION_ID | BIGINT | | The session's identification number. |
| TABLE_PATH | VARCHAR(255) | | Full path of the temporary table. |
| CREATION_TIMESTAMP | TIMESTAMP | | The time that the table was created. |
| TARGET_DATASOURCE_PATH | VARCHAR(2147483647) | | The data source where the temp table data is stored. |

| Column | TDV JDBC Data Type | Nullable | Description |
|-----------------------|---------------------|----------|---|
| TARGET _TABLE_PATH | VARCHAR(2147483647) | | The physical location of the temporary table. |

TRANSACTION_LOG

The TRANSACTION_LOG system table provides a read-only view of the transaction log, which stores transaction states during its lifecycle in case transaction commit fails. You can use log data to recover data manually from a transaction failure. In some cases the system can use this data to complete an interrupted transaction.

Successful transactions are automatically removed from the log upon completion of the commit or rollback operation. Failed transactions remain in the log.

Table view requires the ACCESS_TOOLS and READ_ALL_STATUS rights.

| Column | TDV JDBC Data Type | Nullable | Description |
|--------|--------------------|----------|--|
| TYPE | VARCHAR(28) | | Indicates the type of transaction log entry, which can be: |

| Column | TDV JDBC Data Type | Nullable | Description |
|--|--------------------|----------|--|
| Begin transaction (manual)—Start a transaction supporting manual recovery. | | | |
| Begin transaction (auto)—Start a transaction supporting both manual recovery and automatic compensation. | | | |
| Execute SQL—Execute a SQL statement. | | | |
| Add work unit—Add a work unit (an insert, update, or delete action on a data source). | | | |
| Begin commit | | | |
| End commit | | | |
| Fail commit | | | |
| Begin rollback | | | |
| End rollback | | | |
| Fail rollback | | | |
| Server restart | | | |
| Begin work unit commit | | | |
| End work unit commit | | | |
| Work unit commit failure | | | |
| Work unit commit in doubt | | | |
| Begin work unit rollback | | | |
| End work unit rollback | | | |
| Work unit rollback failure | | | |
| Being work unit compensate | | | |
| End work unit compensate | | | |
| Work unit compensate failure | | | |
| SERIAL | BIGINT | | Unique serial number for the transaction log entry. |
| TIMESTAMP | BIGINT | | The time when the log entry was made, to the millisecond. |
| TRANSACTION_ID | BIGINT | | The unique ID for the transaction to which this log entry applies. |
| WORK_UNIT_ID | BIGINT | Yes | For work unit entries, this is the unique ID; otherwise NULL. |
| MESSAGE | BLOB | Yes | Contains a SQL statement for Execute SQL and Add Work Unit. Contains the exception message for any failure type; otherwise NULL. |

USER_PROFILE

This table provides a list of user profiles.

| Column | TDV JDBC Data Type | Nullable | Description |
|---------------------|--------------------------|----------|--|
| USER_ID | INTEGER | | User Identifier. |
| USER_NAME | VARCHAR | | Name of the user. |
| DOMAIN_NAME | VARCHAR | | Domain for which the user is a member. |
| ATTRIBUTE_NAME | VARCHAR | | Profile attribute. |
| ATTRIBUTE_VALU E | VARCHAR | | Profile value. |

TDV SQL Script

SQL Script is TDV's stored procedure language. It is intended for use in procedural data integration, aggregation, and transformation. It allows conditional logic, looping, and pipelining to be performed in the server. The TDV SQL Script language is similar to the stored procedure languages offered by relational database management systems (RDBMSs).

This topic provides reference to the SQL Script language with several basic examples. It does not provide advanced-level programming tutorials.

Topics for the SQL Script language include:

- [SQL Script Overview, page 345](#)
- [SQL Language Concepts, page 346](#)
- [SQL Script Procedures and Structure, page 362](#)
- [SQL Script Statement Reference, page 371](#)
- [SQL Script Examples, page 406](#)

SQL Script Overview

A SQL Script is a procedure that employs procedure declaration, parameters, statements, variables, data types, procedure calls, SQL keywords, dynamic SQL, conditionals, loops, cursors (simple and streaming), exceptions, and transactions. The following lists the TDV SQL Script keywords.

Procedure Declaration and Parameters

By default (and as required), the procedure name is the same as the name assigned to it in the resource tree.

PROCEDURE; IN, INOUT, OUT

Procedure Call

CALL

Compound Statement

BEGIN/END

Variables

DECLARE can only follow BEGIN.
DECLARE, SET, DEFAULT

Data Types

DECLARE TYPE, BOOLEAN, ROW, XML

Path to a Resource

PATH

SQL Keywords

SELECT INTO, INSERT, UPDATE, DELETE

Dynamic SQL

EXECUTE IMMEDIATE

Conditionals

IF/THEN/ELSE, CASE/WHEN

Loops

LOOP, WHILE, REPEAT/UNTIL, FOR, ITERATE, LEAVE

Cursors

ROW, CURSOR, OPEN, CLOSE, FETCH, SELECT, PIPE (for streaming)

Exceptions

RAISE, EXCEPTION, CURRENT_EXCEPTION

Transactions

TRANSACTION, INDEPENDENT, COMMIT, ROLLBACK

SQL Language Concepts

The following sections cover the basic elements of the SQL Script language.

- [Identifiers, page 347](#)
- [Data Types, page 348](#)
- [Value Expressions, page 352](#)
- [Conditional Expressions, page 353](#)

- [Literal Values, page 354](#)
- [Noncursor Variables, page 354](#)
- [Cursor Variables, page 356](#)
- [Attributes of Cursors, page 356](#)
- [Attributes of CURRENT_EXCEPTION, page 358](#)
- [SQL Script Keywords, page 361](#)

Identifiers

An identifier is a user-defined unique name for an object in SQL Script.

- Identifiers can contain one or more characters.
- Identifiers must begin with an alphabetical character (a-z, A-Z).
- After the initial character, the following characters are valid:
 - Alphanumeric characters: a-z, A-Z, 0-9
 - Separators: , (comma), ; (semicolon), ' ' (pairs of single quotes)
 - Special characters: _ (underscore), / (forward slash), \$ (dollar sign), # (hash symbol)
- An identifier cannot be a SQL Script keyword (see [SQL Script Keywords, page 361](#)), unless the keyword is escaped using double quotes.

Examples of declared variables whose names are SQL Script keywords:

```
DECLARE "VALUE" INTEGER;
DECLARE "CURSOR" CURSOR;
```

Here the SQL Script keywords `VALUE` and `CURSOR` are enclosed in double quotes.

- Escaping an identifier with double quotes also allows it to contain characters that would otherwise not be legal, such as spaces, dashes, or characters from other languages.

Examples of declarations of variables that contain otherwise illegal characters:

```
DECLARE "First Name" VARCHAR(40);
DECLARE "% Returned" DOUBLE;
```

- An identifier can be used for a procedure name, parameter name, cursor name, field name, variable name, cursor variable name, data type name, exception name, or label for a block (such as BEGIN/END, LOOP, WHILE, REPEAT, FOR, LEAVE, ITERATE)

- TDV SQL Script resolves identifiers by a set of processing rules.
 - Identifiers are not case-sensitive.
 - Identifiers within SQL expressions are first evaluated by looking locally in the SQL context. If an identifier is resolved within the local SQL context, the SQL engine does not continue searching.

For example, identifier name matches in database columns in the SQL WHERE clause take precedence over the names of local variables, procedure names, or formal parameters.

- If the identifier is not resolved in the local context, the search proceeds to parent contexts using the smallest prefix basis, moving outward to schema-level scope.
- The SQL context space is not case-sensitive, so differences in capitalization do not distinguish names that match an identifier within the SQL context.
- If no matches are found, an Undeclared Identifier error is returned.

Data Types

TDV supports several data types in SQL Script:

- All of the character strings, numeric, date, time, and TIMESTAMP data types that SQL supports, plus BLOB, CLOB, ROW, and XML. For details, see [Supported Data Types, page 349](#).
- Custom data types. SQL Script lets you declare custom data types for convenience and clarity. You can declare them locally or make them PUBLIC. For details, see [DECLARE TYPE, page 382](#).

The following guidelines apply to TDV data type support:

- References to PUBLIC types must be fully qualified. Such references are valid anywhere the target data type is valid.
- You can use a modifier named PIPE in procedure parameter declarations to pipeline (stream) the output. For details, see [PIPE Modifier, page 364](#).
- After you have declared a custom data type, you can use its name anywhere in the script that you can use a built-in type.
- A PUBLIC type in another procedure can be accessed by specifying the fully qualified path to that procedure, followed by a period, followed by the name of the type.

Supported Data Types

The following table lists all the data types supported in SQL Scripts. All types with optional sizes have default values, as noted.

| Data Type | Range or List of Values |
|---------------------------|--|
| Integer Numeric Types | |
| BIT | 0 or 1 |
| TINYINT | -128 to 127 |
| SMALLINT | -32768 to 32767 |
| INTEGER | -2^{31} to $+2^{31} - 1$ |
| INT | An alias for INTEGER |
| BIGINT | -2^{63} to $+2^{63} - 1$ |
| Non-integer Numeric Types | |
| FLOAT | Approximately 7-digit-precision floating point |
| REAL | An alias for FLOAT |
| DOUBLE | Approximately 17-digit-precision floating point |
| DECIMAL[(p,s)] | Fixed precision number with up to p (precision) digits total and up to s (scale) digits to the right of the decimal point. Default: DECIMAL(32,2). |
| NUMERIC[(p,s)] | Same as DECIMAL, except default is NUMERIC(32,0) |
| Date and Time Types | |
| DATE | |
| TIME | |
| TIMESTAMP | |
| String and Binary Types | |
| CHAR[(n)] | Character string of exactly n characters, padded with spaces. Default for n: 255. |

| Data Type | Range or List of Values |
|----------------------------|---|
| VARCHAR[(n)] Also, CLOB | Unpadded character string of up to n characters. Default for n: 255. |
| BINARY[(n)] | Binary string of exactly n bytes, right-padded as necessary with bytes of zeroes. Default for n: 255. |
| VARBINARY(n) Also, BLOB | Unpadded binary string of up to n bytes. Default for n: 255. |
| Other Types | |
| BOOLEAN | A value of TRUE or FALSE. ('BOOLEAN' is not a valid value.) |
| CURSOR | An untyped cursor (because no list of fields is provided) |
| CURSOR(...) | A cursor defined as a set of fields ('columns') |
| CURSOR(rowType) | A CURSOR declared by referencing a ROW type (instead of specifying fields directly) |
| ROW(...) | A set of fields (also called 'columns') |

| Data Type | Range or List of Values |
|--|---|
| <pre>XML [({ DOCUMENT CONTENT SEQUENCE } [(ANY UNTYPED XMLSCHEMA schema-details)])]</pre> <p>schema-details:</p> <pre>URI target-namespace-uri [LOCATION schema-location] [{ ELEMENT element-name NAMESPACE namespace-uri [ELEMENT element-name]}] NO NAMESPACE [LOCATION schema-location] [{ ELEMENT element-name NAMESPACE namespace-uri [ELEMENT element-name]}]</pre> | <p>An XMLvalue. Default: 'No Schema.'</p> <ul style="list-style-type: none">target-namespace-uri: a string literal that represents a valid URIschema-location: a string literal that represents a valid URInamespace-uri: a string literal that represents a valid URIelement-name: any valid identifier |

Example (Declaring a Custom Data Type)

You can declare a custom data type in SQL Script for later referencing:

```
DECLARE TYPE SocialSecurityType VARCHAR(12);
DECLARE ssn SocialSecurityType;
DECLARE data ROW (name VARCHAR(40), ssn SocialSecurityType);
```

Example (Referencing a Custom Data Type)

If you have declared a custom data type in SQL Script named SocialSecurityType in a procedure named TypeSample in the folder /shared/examples, you can reference the type as follows:

```
DECLARE ssn /shared/examples/TypeSample.SocialSecurityType;
```

Example (XML Data Type)

You can declare an XML data type in SQL Script as follows:

```
cast('<item> </item>' as XML (SEQUENCE))
cast('<bar></bar>' as XML(SEQUENCE(ANY)))
PROCEDURE item()
BEGIN
DECLARE item
XML (SEQUENCE (XMLSCHEMA URI LOCATION 'http://www.w3.org/2001/
XMLSchema-instance' [^] ELEMENT xsi));
END
```

Value Expressions

A value expression in a SQL Script is anything that resolves to a value.

Syntax

The syntax for a value expression is identical to a projection in a SELECT statement, except that instead of using column names you can use variable names in a value expression.

Remarks

- Cursor variables cannot be used in a value expression by themselves, although attributes of cursor variables can be used. See [DECLARE CURSOR of Type Variable, page 378](#) for information on declaring cursor variables, and [Attributes of Cursors, page 356](#) for information on cursor attributes.
- The keyword CURRENT_EXCEPTION cannot be used in a value expression by itself, although attributes of it can be used. For details, see [Attributes of CURRENT_EXCEPTION, page 358](#).

Errors

The following table describes the errors that can occur while resolving a value expression.

| Error Message | Cause |
|------------------------------------|--|
| Undefined variable | An identifier is encountered that is not defined in the current scope. |
| Incorrect use of a cursor | A cursor is used in a value expression. |
| Incorrect use of CURRENT_EXCEPTION | The keyword CURRENT_EXCEPTION is used in a value expression. |

Conditional Expressions

A conditional expression in a SQL Script is anything that resolves to a boolean value.

Syntax

The syntax for a conditional expression is identical to what you can use as a WHERE clause, except that instead of using column names you use variable names in a conditional expression.

Remarks

- Cursor variables can be used in a conditional expression only with the keyword IS NULL or IS NOT NULL. Cursor variables cannot be used in other conditional expressions, although attributes of cursor variables can be used. See [DECLARE CURSOR of Type Variable, page 378](#), for information on declaring cursor variables, and [Attributes of Cursors, page 356](#), for information on cursor attributes.
- A boolean variable or literal can be used as a condition. See [Literal Values, page 354](#), for information on declaring literals.
- The keyword CURRENT_EXCEPTION cannot be used in a conditional expression by itself, although attributes of it can be used. For details, see [Attributes of CURRENT_EXCEPTION, page 358](#).

Errors

The following table describes the errors that can occur while resolving a conditional expression.

| Error Message | Cause |
|------------------------------------|--|
| Undefined variable | An identifier is encountered that is not defined in the current scope. |
| Incorrect use of a cursor | A cursor is used in a conditional expression with something other than IS NULL or IS NOT NULL. |
| Incorrect use of CURRENT_EXCEPTION | The keyword CURRENT_EXCEPTION is used in a conditional expression. |

Literal Values

A SQL Script can contain any literal value that is valid in SQL, plus type ROW or XML (which need to be defined).

Syntax (ROW-Type Literal Value)

ROW(<valueExpression>, ...)

Syntax (XML-Type Literal Value)

There is no literal format for an XML type. Use the following syntax to create an XML type.

CAST ('xml_string' AS XML)

Remarks

- The symbols TRUE and FALSE are reserved for use as literal boolean values.
- Literal values are delimited by single quotes ('string'). To specify an apostrophe within a string, use two apostrophes in a row (').
- There is no literal format for a cursor type. For details, see [DECLARE CURSOR of Type Variable, page 378](#).

Noncursor Variables

Noncursor variables in SQL Script are expressions or other elements that resolve to single values. You can define a noncursor variable by specifying its name and data type.

Syntax

DECLARE <varName>[,...] <dataType>
[DEFAULT <valueExpression>]

Remarks

- The DEFAULT syntax is optional. It is used to initialize a variable.
- Any variable that is not initialized with a DEFAULT clause has the value NULL.
- Variables can be used in SQL Script expressions anywhere a literal value is valid. For example, both 1 + 1 and x + y are valid expressions (assuming x and y are declared variables).
- Variables in SQL Scripts are subject to scoping rules.

- A variable can be declared within a block that has the same name as a variable in a parent block. Parameters are treated as if they were defined in the main block of the procedure.
- String-type variables are delimited by single quotes ('string'). To specify an apostrophe within a string, use two apostrophes in a row ("').
- You can declare variables, parameters, and column definitions that are of type BLOB or CLOB.
- You can declare multiple variables at one time, provided all the variables are of the same data type and each has a unique name.
- The <valueExpression> can use IN parameters, previously declared variables in this block, and any variables in parent blocks. In the current block, the value expression cannot use variables that are defined later. If the value expression's type does not match the variable's type, an implicit cast is performed (if possible). For information about IN parameters, see [SQL Script Procedure Header, page 363](#).
- If the evaluation of the value expression causes an exception, any other declared variables that have not yet been initialized are set to NULL before entering the exception handler.

Examples

```
PROCEDURE p ( )
BEGIN
  DECLARE a INTEGER;
  DECLARE b DATE;
  DECLARE c TIME;
  DECLARE d TIMESTAMP;
  DECLARE e DECIMAL;
  DECLARE f FLOAT;
  DECLARE g VARCHAR;
  DECLARE h CHAR;
END

PROCEDURE p ( )
BEGIN
  DECLARE x INTEGER;

  SET x = 1;
  DECLARE x INTEGER; --illegal
END
```

Cursor Variables

Cursor variables in SQL Script are expressions or other elements that resolve to cursors. You can define a cursor variable by providing a unique name and optionally specifying its data type, as described in [DECLARE CURSOR of Type Variable, page 378](#).

Syntax

```
DECLARE <varName> CURSOR
[<dataType>]
```

Remarks

- The optional <dataType> can be a named ROW data type, or the syntax for a ROW data type.
- The syntax for a ROW data type is: <colName> <dataType> [...].
- There are no attributes on a ROW variable.
- You access a row using rowVar.columnName to get a column.
- When declared, cursor variables are initialized to NULL. They cannot be initialized to any other value at declaration.
- A cursor variable with a type can be assigned from any cursor with the same ROW type, or to any cursor variable with the same ROW type.
- A cursor variable without a type can be assigned from any cursor, or to any cursor. Assigning to a typed cursor forces a run-time schema match comparison and raises an exception on a mismatch.
- Assigning a cursor creates a reference to the original cursor's state. This means that opening, closing, or fetching from the original cursor or the variable has the same effect, and alters what the other would see.
- For further information, see [Attributes of Cursors, page 356](#), [OPEN, page 399](#), [FETCH, page 392](#), and [CLOSE, page 375](#).

Attributes of Cursors

You can obtain the attributes of a cursor in SQL Script. See [DECLARE CURSOR of Type Variable, page 378](#), [OPEN, page 399](#), [FETCH, page 392](#), and [CLOSE, page 375](#) for details about cursors.

Syntax

```
<cursor>.<attribute>
```


Remarks

The following table describes cursor attributes

| Attribute | Description |
|-----------|---|
| ISOPEN | A boolean that indicates whether the cursor is open or not. |
| ROWTYPE | The ROW data type for the cursor. NULL for an untyped cursor. |
| ROWCOUNT | Number of rows fetched from the cursor if it is open. NULL if it is not open. |
| FOUND | A boolean that is true if the last fetch from the cursor found a row. NULL if not open, or open and not fetched from. |

Example

The following example returns the n^{th} value of a cursor of VARCHARs.

```
PROCEDURE nth (IN n INTEGER, IN crs CURSOR(name VARCHAR), OUT name VARCHAR)
a_lab:
BEGIN
IF NOT crs.ISOPEN THEN
OPEN crs;
END IF;
LOOP
FETCH crs INTO name;
IF NOT crs.FOUND OR nth >= crs.ROWCOUNT THEN
LEAVE a_lab;
END IF;
END LOOP;
CLOSE crs;
END
```

The following example makes use of the ROWTYPE attribute:

```
CURSOR m1 IS
SELECT last_name, hire_date, job_id
FROM employees
WHERE employee_id = 5446;
employee_rec m1%ROWTYPE;

BEGIN
OPEN m1;
FETCH m1 INTO employee_rec;
DBMS_OUTPUT.PUT_LINE('Employee name: ' || employee_rec.last_name);
END;
```

Attributes of CURRENT_EXCEPTION

In SQL Script, you can obtain the attributes of an exception while within the exception handler.

For details, also see:

- [SQL Script Exceptions, page 359](#)
- [Raising and Handling Exceptions, page 369](#)
- [External Exceptions, page 371](#)
- [DECLARE EXCEPTION, page 381](#)

Syntax

CURRENT_EXCEPTION.<attribute>

Remarks

The following table describes cursor exception attributes.

| Attribute | Description |
|-----------|--|
| NAME | <p>A string that is the exception’s name. This name is fully qualified, as follows:</p> <p>/ns1/ns2/procedure.s1.s2.exceptionName</p> <p>The ns1 and ns2 are namespace elements of the path. The s1 and s2 are compound statement blocks and are either named according to the label on that block or as unnamed# where # is an integer value.</p> |
| ID | <p>An integer that is the exception’s system ID. All user exceptions have the ID -1 (negative one). System exceptions all have unique IDs.</p> |
| MESSAGE | <p>The VARCHAR(255) value defined for the current exception. If no value is defined for the exception, then this attribute is NULL.</p> |
| TRACE | <p>The VARCHAR(32768) value defined contains the exception stack trace as a string.</p> |

If the exception handler includes a compound statement, CURRENT_EXCEPTION within the BEGIN portion refers to the current exception of the parent scope, but within the exception handler portion of the child scope CURRENT_EXCEPTION refers to the local exception and there is no way to access the parent exception. For details, see [Compound Statements, page 365](#).

Example

PROCEDURE p (IN x INTEGER, OUT result VARCHAR)

```

BEGIN
  CALL /shared/f(x);
EXCEPTION
ELSE
  IF CURRENT_EXCEPTION.MESSAGE IS NOT NULL THEN
    SET result = CURRENT_EXCEPTION.MESSAGE;
  ELSE
    SET result = CURRENT_EXCEPTION.NAME;
  END
END

MESSAGE:  'x must be > 0. x = -123'
NAME:     '/shared/f.illegal_arg_ex'

```

SQL Script Exceptions

The following is a list of SQL Script exceptions that can be thrown. The message that is passed is left to the author of the SQL Script.

| Exception Message | Description |
|------------------------------|---|
| CannotExecuteSelectException | An attempt is made to execute a SELECT statement. SELECT statements are opened, not executed. INSERT, UPDATE, and DELETE statements are executed. |
| CannotOpenCursorException | An attempt is made to open a cursor that is either a NULL reference variable, or is a cursor that is not defined within the current procedure that has already been closed. |
| CannotOpenNonSelectException | An attempt is made to open an INSERT, UPDATE, or DELETE statement. INSERT, UPDATE, and DELETE statements are executed, not opened. SELECT statements are opened. |
| CursorAlreadyOpenException | An attempt is made to open a cursor that is already open. |
| CursorNotOpenException | An attempt is made to fetch from or to close a cursor that is closed, or to insert into or close a PIPE that is closed. |
| CursorTypeMismatchException | An attempt is made to open a cursor using dynamic SQL and the projections from the SQL do not match the cursor's type definition. |
| DuplicateNameException | An attempt is made to name something and that name is already in use. |
| EvaluationException | An error is encountered evaluating an expression. |
| IllegalArgumentException | An argument is passed into a procedure with an illegal value. |

| Exception Message | Description |
|---------------------------------|---|
| IllegalStateException | A procedure cannot perform its task due to some unexpected state. |
| NotAllowedException | An attempt is made to perform a task that is not allowed due to policy restrictions or other limitations. |
| NotFoundException | An attempt is made to use a resource or other item that does not exist. |
| NotSupportedException | An attempt is made to use a feature that is not supported. |
| NullVariableException | An attempt is made to access a data member of a NULL variable. For example, to access a data member of a ROW variable that is currently NULL. |
| ParseException | A dynamic SQL statement fails to parse or resolve correctly. This can be due to a syntax error or a reference to a nonexistent column, table, procedure, or function. |
| PipeNotOpenException | An attempt is made to insert into or to close a PIPE that is already closed. |
| ProcedureClosedException | A procedure is closed forcibly by the system due to being aborted by the caller or an administrator. |
| ProtocolException | A task fails due to a processing error on a data protocol. |
| SecurityException | An attempt is made to perform an action without proper privileges. |
| SystemException | A general failure in the runtime is encountered |
| TransactionClosedException | An attempt is made to perform a transactional task (such as fetching from a cursor) after the transaction has been committed or rolled back. |
| TransactionFailureException | A transaction failure occurs. |
| UnexpectedRowCountException | A cursor has an unexpected number of rows returned. For example, the SELECT INTO statement requires the cursor to return exactly one row. |
| UnopenedCursorReturnedException | An unopened cursor is returned from a procedure. Cursors must be NULL or be open when returned. |

| Exception Message | Description |
|--------------------|--|
| SOAPFaultException | A SOAP Fault is returned from a Web service. |

SQL Script Keywords

SQL Script keywords are the character strings that SQL Script treats as reserved words.

Note: TDV does not treat all SQL-99 reserved words as SQL Script keywords.

SQL Script keywords are not case-sensitive. However, TDV documentation uses uppercase letters to distinguish keywords from other words.

Although it is not recommended, you can use SQL Script keywords in roles other than their intended syntax, as long as you set them off in double quotes. For example:

```
SELECT "BEGIN" INTO ...
```

The following table lists the SQL Script keywords.

| SQL Script Keywords | | | |
|---------------------|-----------------------|-------------|----------|
| AS | BEGIN | CALL | CASE |
| CAST | CLOSE | COMMIT | CREATE |
| DROP | CURRENT_EXC EPTION | CURSOR | DO |
| DECLARE | DEFAULT | DELETE | ELSE |
| ELSE IF | END | EXCEPTION | EXECUTE |
| FALSE | FETCH | FOR | IF |
| IMMEDIATE | IN | INDEPENDENT | INOUT |
| INSERT INTO | INTO | ITERATE | LEAVE |
| LOOP | OPEN | OUT | PIPE |
| PROCEDURE | PUBLIC | RAISE | REPEAT |
| ROLLBACK | ROW | SELECT | SET |
| THEN | TRANSACTION | TRUE | TRUNCATE |

| SQL Script Keywords | | | |
|---------------------|-------|--------|-------|
| TYPE | UNTIL | UPDATE | VALUE |
| WHEN | WHILE | | |

SQL Script Procedures and Structure

The following sections cover the syntactic details of a procedure.

- [Basic Structure of a SQL Script Procedure, page 362](#)
- [SQL Script Procedure Header, page 363](#)
- [Compound Statements, page 365](#)
- [Independent Transactions, page 366](#)
- [Compensating Transactions, page 368](#)
- [Exceptions, page 369](#)

Basic Structure of a SQL Script Procedure

The basic structure of a SQL Script procedure begins with the word `PROCEDURE`, followed by the name of the procedure, an open parenthesis, and a closed parenthesis. Next is a block that begins with the word `BEGIN` and ends with the word `END`. The code for the procedure is placed between the `BEGIN` and `END` statements.

Syntax

```
PROCEDURE myProcedure()  
  BEGIN  
    -- Add your code here  
  END
```

Commenting SQL Script Code

A line that begins with two dashes (`--`) is a comment (annotation) line. Comment lines are not executed.

Another way of commenting, similar to the style followed in Java programming, is shown in the following example:

```
PROCEDURE myProc2()  
  BEGIN  
    /*  
     * This is a multiline comment
```

```

*/
DECLARE x INTEGER; -- This is a comment
CALL /shared/procedures/aProcedure(x /* param1 */);
END

```

SQL Script Statement Delimiter

The statement delimiter is a semicolon (;).

SQL Script Procedure Header

A procedure declaration in SQL Script defines the input parameters and output parameters of the procedure. To call a procedure, see [CALL, page 373](#).

Syntax

```

PROCEDURE <procedureName> ( [<parameterList>] )
<statement>

```

The parentheses in the procedure's syntax are optional. If there are parentheses, they can be empty or they can contain a list of parameters.

Remarks

- A parameter list ([<paramList>](#)) is a comma-separated list of parameters of the form:
{ IN | INOUT | OUT } <parameterName> <dataType>
- The data type of a parameter ([<dataType>](#)) can be any type listed in [Data Types, page 348](#), except ROW.
- You can use any PUBLIC data type defined in the main compound statement within the procedure declaration (indicated by <compoundStatement> in the syntax for a procedure). This way a parameter can be defined to be of a named type instead of always being primitive.

Examples

```

PROCEDURE init_table (IN employee_id INTEGER)
BEGIN
  INSERT INTO T (empid) VALUES (employee_id);
END
PROCEDURE cur_month (OUT x INTEGER)
BEGIN
  SET x = MONTH (CURRENT_DATE());
END
PROCEDURE inc (INOUT x INTEGER)
BEGIN
  SET x = x + 1;

```

```

END
PROCEDURE inc (IN x INTEGER)
BEGIN
    SET x = 5; -- Error
END

```

PIPE Modifier

A modifier named PIPE is used in SQL Script for streaming a cursor. It can be used only in procedure parameter declarations, and its purpose is to pipeline the output.

Syntax

```

IN <parameterName> PIPE <cursorDataType>
OUT <parameterName> PIPE <cursorDataType>

```

Remarks

- The PIPE modifier can be applied to any IN or OUT cursor data type.
- The PIPE modifier cannot be used on INOUT parameters or on any noncursor data type.
- An IN parameter with the PIPE modifier can be passed any PIPE variable that comes from an IN or OUT parameter of the current procedure.
- An OUT parameter with the PIPE modifier must be passed a cursor variable with the same schema as the PIPE.
- Within a PROCEDURE, a PIPE variable (either IN or OUT) can be used in INSERT statements. For details, see [INSERT, page 396](#).
- Procedures with a PIPE modifier on an IN parameter do not run in a separate thread.
- Any procedure with the PIPE modifier on an OUT parameter runs in a separate thread. The calling procedure continues execution as soon as the pipelined procedure begins execution. The calling procedure finds the OUT cursor already initialized, and opens the cursor and can fetch from it. (For details, see [FETCH, page 392](#).) If the calling procedure accesses any non-PIPE OUT parameter, however, the calling procedure blocks until the pipelined procedure ends execution. This is because the final values of non-PIPE outputs are not known until the procedure completes.
- A PIPE modifier can be in an INSERT statement within an EXECUTE IMMEDIATE statement.

Example

The following procedure returns a cursor with all of the names reversed.

```
PROCEDURE reverse_all (OUT result PIPE (rev_name VARCHAR))
BEGIN
  DECLARE c CURSOR FOR SELECT name FROM /shared/T;
  DECLARE name VARCHAR;
  OPEN c;
  REPEAT
    FETCH c INTO name;
    CALL /shared/reverse(name, name);
    INSERT INTO result (rev_name) VALUES (name);
  UNTIL NOT c.FOUND
  END REPEAT;
END
```

Compound Statements

A compound statement in SQL Script has multiple statements within a BEGIN-END pair. A compound statement must end with a semicolon if it is not the root statement.

Syntax

```
[<label>:]
BEGIN
[<transactionSpecification>]
[<declaration>; ...]
[<statement>; ...]
[<exceptionBlock>]
END [<label>]
```

Remarks

- The label is for use with the LEAVE statement defined in [LEAVE, page 397](#).
- The label is an optional identifier used to name the block. The root BEGIN statement (the one directly following the PROCEDURE declaration) can have (be preceded by) a label.
- When **BEGIN** is present, **END** is optional. If **BEGIN** is not present, it is illegal to have an **END** label. If both **BEGIN** and **END** are present, both must have the same identifier.
- A compound statement can be empty.

Example

```
PROCEDURE init_table()
BEGIN
  DELETE FROM T;
  INSERT INTO T DEFAULT VALUEs;
```

END

Independent Transactions

An independent transaction in SQL Script is a set of work that can be rolled back or committed on its own, regardless of what happens to the main transaction.

Syntax

INDEPENDENT [<option> ...] TRANSACTION

Remarks

- Options (<option> ...) are not case-sensitive.
- The following table describes the option flags for an independent transaction.

| Option Flag | Significance |
|-----------------------------------|--|
| ROLLBACK_ON_FAILURE BEST_EFFORT | <p>This pair of flags indicates whether the transaction should be rolled back if a failure occurs during COMMIT (ROLLBACK_ON_FAILURE, the default) or not (BEST_EFFORT). You cannot set both of these flags at the same time.</p> <p>With ROLLBACK_ON_FAILURE, failure to commit any part of the transaction causes uncommitted parts to be discarded, and causes already committed parts to be compensated (according to the COMPENSATE/NOCOMPENSATE option).</p> <p>With BEST_EFFORT, even if one part of the transaction cannot be committed, as many other parts as possible are still committed. The failed parts are logged.</p> |
| COMPENSATE NOCOMPENSATE | <p>This pair of flags indicates whether the compensation blocks should be run if the transaction rolls back (COMPENSATE, the default) or not (NOCOMPENSATE). You cannot set both of these flags at the same time.</p> <p>NOCOMPENSATE improves performance at the risk of compensation. However, setting this to COMPENSATE has no performance cost unless you define a compensation block.</p> |

| Option Flag | Significance |
|------------------|---|
| IGNORE_INTERRUPT | <p>This group of flags indicates what the system should do if the server goes down or is interrupted when the transaction commit is partially complete. You cannot set more than one of these flags at a time.</p> <ul style="list-style-type: none"> IGNORE_INTERRUPT (the default) causes the server to take no special action on restart. LOG_INTERRUPT causes the server to store basic transaction information before beginning to commit so that on restart it can detect any transactions in progress and log their failure. This option requires two meta-commits per transaction (start and stop). FAIL_INTERRUPT causes the server to store enough information to perform the requested failure model upon server startup for any in-progress transactions. This option is expensive, because it requests meta-commits for start of transaction, for end of transaction, and between each pair of sources it commits to. |
| LOG_INTERRUPT | |
| FAIL_INTERRUPT | |

- The BEGIN statement can be followed by a transaction specifier. (See [Compound Statements, page 365](#) for information on using BEGIN in a compound statement.) If there is no specifier, the block runs within its parent’s transaction, and any work it performs is part of the parent transaction.
- When a compound statement is declared as having an independent transaction, all actions in that scope are part of the transaction. See [Compound Statements, page 365](#) for information on declaring a compound statement.
- Calling COMMIT is recommended but not required. See [COMMIT, page 376](#).
- A normal exit from the scope commits the transaction.
- Exiting the scope through an unhandled exception causes a transaction rollback.
- Exiting through any handled exception does not implicitly roll back the transaction. You must explicitly roll back the transaction if that is what you want. See [ROLLBACK, page 402](#).

Example

You can use the BEST_EFFORT and NOCOMPENSATE options as follows in SQL Script:

```
PROCEDURE myProcedure ()
BEGIN INDEPENDENT BEST_EFFORT NOCOMPENSATE TRANSACTION
-- Add your code here
END
```

Error

The following table describes the error that can occur while resolving a transaction.

| Error Message | Cause |
|---------------------|--|
| Conflicting options | Two mutually exclusive options have been declared. |

Compensating Transactions

A compensating transaction in SQL Script is a special handler that a COMPENSATE exception invokes to restore transactional integrity after a compound statement ends.

Remarks

- The presence of a handler for the COMPENSATE exception causes special behavior at run time. Unlike other exceptions, this exception cannot be handled by an ELSE clause; it can only be handled explicitly.
- The COMPENSATE exception is special because it is the only exception that can be raised after the compound statement ends. It can be called a long time after the statement ends. This exception is raised if the transaction is rolled back either explicitly by the transaction’s controller or by the system, if a failure occurs during commit.
- The COMPENSATE handler has access to all the variables that the block can see, like other exception handlers. This is a copy of those variables at the time the block exited.
- Compensation can be expensive because this additional storage of variable state has to be kept for every execution of the block. For example, if the block occurs in a loop that ran 1,000 times, 1,000 separate compensation states need to run. For this reason, monitor the COMPENSATE handler carefully.
- Only the current local data state is preserved for the handler. The global system state is not preserved. That is, if you call another procedure, it cannot be in the same state as it was the first time this block was run. For this reason, any required state should be captured during the normal run into variables so they can be used during the COMPENSATE handler.

Examples

```
PROCEDURE p ()  
BEGIN INDEPENDENT TRANSACTION  
  <statement>  
END
```

The insert is automatically committed in the example below.

```
PROCEDURE p ()
BEGIN INDEPENDENT TRANSACTION
  INSERT INTO /shared/T (name, score) VALUES ('Joe', 123);
END
```

The insert is automatically rolled back in the example below.

```
PROCEDURE p ()
BEGIN INDEPENDENT TRANSACTION
  DECLARE my_exc EXCEPTION;
  INSERT INTO /shared/T (name, score) VALUES ('Joe', 123);
  RAISE my_exc;
END
```

The insert is automatically committed in the example below.

```
PROCEDURE p ()
BEGIN INDEPENDENT TRANSACTION
  DECLARE my_exc EXCEPTION;
  INSERT INTO /shared/T (name, score) VALUES ('Joe', 123);
  RAISE my_exc;
EXCEPTION
  ELSE
END
```

Exceptions

You can define exceptions in SQL Script by providing a unique name for the exception and defining a procedure of that name to handle the exception condition.

- [Attributes of CURRENT_EXCEPTION, page 358](#)
- [Raising and Handling Exceptions, page 369](#)
- [External Exceptions, page 371](#)

Syntax

```
DECLARE [PUBLIC] <exceptionName> EXCEPTION
```

You can declare an exception in a child scope that has the same name as the one declared in the parent scope. If you do that, the one in the parent scope is not visible within the child scope.

Raising and Handling Exceptions

A BEGIN/END block in SQL Script can have an optional exception section.

Syntax

```

BEGIN
... ..
EXCEPTION
[WHEN <exceptionName>
[OR <exceptionName> ...]
THEN <statements> ...]
[ELSE <statements>]
END

```

Remarks

- If the EXCEPTION block is declared, it must contain at least one WHEN or one ELSE clause. An EXCEPTION block can contain any number of WHEN clauses, but only one ELSE clause.
- When an exception is raised in a BEGIN/END block, the first exception-handler WHEN clause that matches the exception is executed.
- All variables from the scope are available within the exception handler. This technique is different from Java, for example. In Java, nothing from the TRY block is available in the CATCH block. In SQL Script, all variables available within the BEGIN area are available within the EXCEPTION area. They do not go out of scope until END is reached.
- If an exception is not handled within a block, that block leaves scope as with a LEAVE statement and the same exception is raised in the parent scope, where it can be handled. If there are no further scopes, the exception is thrown out of the procedure to the caller. If the caller is SQL Script, SQL Script receives this error. If the caller is JDBC or a Java Procedure, a Java exception is received.

If the caller is in a SQL FROM clause, the statements ends with a runtime exception.

- Any exception raised while in an exception handler, immediately leaves the current scope as if it were an unhandled exception in this scope.
- Use the RAISE statement to raise an exception again.

Example

```

PROCEDURE p (IN x INTEGER, OUT result BIT)
BEGIN
  DECLARE illegal_arg_ex EXCEPTION;
  ...
  IF x < 0 THEN
    RAISE illegal_arg_ex;
  END
  SET result = 1;  --success
EXCEPTION
  WHEN illegal_arg_ex THEN
    SET result = 0;  --failure

```

END

External Exceptions

System exceptions in SQL Script are considered to be globally reserved names, but they can be referenced by SQL Script procedures. If a user-defined exception is made public, it can be used by other procedures.

Syntax

<compNamespacePath>.<exceptionName>

Remarks

- You can invoke a system exception or other public exceptions from a SQL Script procedure by including a TDV namespace path (<compNamespacePath>) followed by a dot and the exception name (<exceptionName>) in the script.
- You can view the system exceptions available to SQL Script procedures on the Exceptions tab of /lib/util/System in Studio.

Example

/lib/util/System.NotFoundException

SQL Script Statement Reference

The following table lists all the SQL Script statements discussed in detail.

| Statement | Statement |
|--|-----------------------------------|
| BEGIN...END, page 372 | FETCH, page 392 |
| CALL, page 373 | FOR, page 394 |
| CASE, page 374 | IF, page 395 |
| CLOSE, page 375 | INSERT, page 396 |
| COMMIT, page 376 | ITERATE, page 397 |
| CREATE TABLE, page 376 | LEAVE, page 397 |
| CREATE TABLE AS SELECT, page 377 | LOOP, page 398 |

| Statement | Statement |
|--|--|
| CREATE INDEX , page 377 | OPEN , page 399 |
| DECLARE Constants , page 378 | PATH , page 400 |
| DECLARE CURSOR of Type Variable , page 378 | RAISE , page 401 |
| DECLARE EXCEPTION , page 381 | REPEAT , page 402 |
| DECLARE TYPE , page 382 | ROLLBACK , page 402 |
| DECLARE Variable , page 383 | SELECT INTO , page 403 |
| DECLARE VECTOR , page 384 | SET , page 404 |
| DELETE , page 390 | TOP , page 404 |
| DROP TABLE , page 391 | UPDATE , page 405 |
| EXECUTE IMMEDIATE , page 391 | WHILE , page 406 |
| FIND_INDEX , page 392 | |

BEGIN...END

BEGIN and END enclose a SQL Script procedure, which can include one statement or multiple statements (that is, a compound statement).

Syntax

```
[<label>:]  
BEGIN  
[<transactionSpecification>  
[<declaration>; ...]  
[<statement>; ...]  
[<exceptionBlock>  
END [<label>]
```

Remarks

- The order of the parameters in the procedure’s declaration is important. While it is conventional to list IN, then INOUT, then OUT parameters in that order, they can be intermixed.
- IN parameters are unchangeable in the procedure (like a const parameter).

- OUT parameters are initialized to NULL within the procedure. Setting a value into an OUT parameter assigns the value to the variable in the caller.
- INOUT parameters are like OUT parameters that are pre-initialized by the caller. Any calling environment that does not have variables should treat these parameters as if they were a pair of IN and OUT parameters.

CALL

The **CALL** statement is used to call a procedure in SQL Script.

Syntax

```
CALL <procedureName> ( [<valueExpression>[,...]] )
```

The <procedureName> refers to the name of a procedure declared using the syntax for a procedure declaration. See [SQL Script Procedure Header, page 363](#) for procedure declaration.

Parentheses in the CALL syntax are not required if there are no parameters.

Remarks

- IN parameters can be passed any value expression. For details, see [Value Expressions, page 352](#). The expression is implicitly cast, if required, to match the type of the IN parameter. IN parameters can be literals, expressions, or variables. If an IN parameter is a variable, the value is not altered. IN parameters with the PIPE modifier ([PIPE Modifier, page 364](#)) can only pass in variables that are also PIPE variables. This means only IN or OUT parameters of the current procedure that have the PIPE modifier can be passed in.
- The expressions being passed to IN parameters are evaluated from left to right.
- INOUT and OUT parameters must be passed a variable of the appropriate type. No implicit type conversion is supported. For INOUT parameters, the value is not altered if it is not changed in the procedure. For OUT parameters, the value is set to NULL if not altered in the procedure. OUT parameters with the PIPE modifier can only be passed a cursor variable with the same cursor type as the PIPE.

Examples

```
PROCEDURE square (IN x INTEGER, OUT result INTEGER)
BEGIN
  SET result = x * x;
END
```

```

PROCEDURE p( )
BEGIN
  DECLARE y INTEGER;
  CALL square(2, y);
  -- y is 4
  CALL sqaure(y, y);
  -- y is 16
END

```

```

PROCEDURE factorial (IN x INTEGER, OUT result INTEGER)
BEGIN
  IF x = 1 THEN
    SET result = 1;
  ELSE
    CALL /shared/factorial(x-1; result);
    SET result = x * result;
  END IF
END

```

CASE

A CASE statement in SQL Script evaluates a list of conditions and returns one of multiple possible result expressions. The CASE statement has two valid formats.

Syntax 1

Use the <valueExpression> syntax to evaluate an expression once and then find a matching value. The WHEN clauses are evaluated in order and the first match is used.

```

CASE <valueExpression>
WHEN <valueExpression> THEN <statements>
[...]
[ELSE <statements>]
END AS <new_column_name>

```

Syntax 2

Use the <conditionalExpression> syntax to evaluate a series of tests like an IF/THEN/ELSEIF/ELSE. The WHEN clauses are evaluated in order and the first match is used.

```

CASE
WHEN <conditionalExpression> THEN <statements>
[...]
[ELSE <statements>]
END AS <new_column_name>

```

Remark

There can be zero or more statements in the area indicated by <statements>.

Examples

```

PROCEDURE get_month_name(OUT month_name VARCHAR)
BEGIN
  CASE MONTH(CURRENT_DATE())
    WHEN 1 THEN
      SET month_name = 'JAN';
    WHEN 2 THEN
      SET month_name = 'FEB';
    WHEN 3 THEN
      SET month_name = 'MAR';
    ...
    WHEN 11 THEN
      SET month_name = 'NOV';
    WHEN 12 THEN
      SET month_name = 'DEC';
  END CASE;
END

PROCEDURE get_duration(IN seconds INTEGER, OUT result VARCHAR)
BEGIN
  CASE
    WHEN seconds < 60 THEN
      SET result = CAST (
        CONCAT(seconds, ' seconds') AS VARCHAR);
    WHEN seconds < 60*60 THEN
      SET result = CAST (
        CONCAT(seconds/60, ' minutes') AS VARCHAR);
    ELSE
      SET result = CAST (
        CONCAT(seconds/3600, ' hours') AS VARCHAR);

  END CASE;
END

```

CLOSE

The CLOSE statement in SQL Script is used to close a cursor. See [DECLARE CURSOR of Type Variable, page 378](#) for details on declaring cursors.

Syntax

```
CLOSE <cursor>
```

Errors

The following table describes the errors that can occur while executing a CLOSE statement.

| Error Message | Cause |
|----------------------|--|
| Uninitialized cursor | A cursor variable is used and is not initialized at the time it is opened. |
| Cursor is not open | CLOSE was invoked when the cursor was not open. |

COMMIT

The COMMIT statement in SQL Script is used to commit an independent transaction inside a compound statement.

Syntax

COMMIT

Remark

- It is illegal to call COMMIT in a compound statement that is not declared INDEPENDENT.
- For details, see [Independent Transactions, page 366](#), [Compensating Transactions, page 368](#), and [Compound Statements, page 365](#).

Example

```
PROCEDURE p ()
BEGIN INDEPENDENT TRANSACTION
  DECLARE my_exec EXCEPTION;
  INSERT INTO /shared/T (name, score) VALUES ('Joe', 123);
  COMMIT;
  RAISE my_exec;
END
```

CREATE TABLE

Creates a new table in the database.

Syntax

```
CREATE TABLE table_name (
  column1 datatype,
  column2 datatype,
  column3 datatype,...
);
```

CREATE TABLE AS SELECT

Create a table from an existing table by copying the existing table's columns. The new table is populated with the records from the existing table.

Creates a TEMPORARY table as a copy of an existing table.

Syntax

```
CREATE [TEMPORARY] TABLE table-name AS QUERY_EXPRESSION
```

```
CREATE [TEMPORARY] TABLE new_table
AS (SELECT * FROM old_table);
```

Remarks

- The QUERY_EXPRESSION can be any select query without an ORDER BY or LIMIT clause.
- The temporary table will be empty on first access, can optionally be returned to empty state at every COMMIT by using the ON COMMIT clause. The temporary tables are automatically cleaned up by the server at the end of the user session. You can also explicitly drop them if needed in between the session.
- If most of the queries are going against a particular database, the performance of the joins on temporary table with the persisted table might be better with a specific temporary table storage location. The privileges associated with the Temporary Table Container affect the user who can create and use temporary tables if the DDL Container is set. The temporary table storage location can be changed by editing the Temporary Table Container configuration parameter through Studio.

Examples

```
CREATE TABLE queenbee
AS (SELECT * FROM babybee);
```

OR

```
CREATE TEMPORARY TABLE queenbee
AS (SELECT * FROM babybee);
```

CREATE INDEX

Creates indexes in the table.

Syntax

```
CREATE INDEX index_name
ON table_name (column1, column2, ...);
```

Example

```
CREATE INDEX index_1
ON queenbee (column_bee1)
```

DECLARE Constants

You can define constants in SQL Script by declaring them with unique names.

Syntax

```
DECLARE [PUBLIC] <variableName>[,...] <type> DEFAULT <valueExpression>]
```

Remarks

- You must declare a CONSTANT before using it.
- DEFAULT initializes the variable.
- If you declare multiple variables (for example, ROW (a INT, b CHAR)), enclose a comma-separated list of default values in parentheses in the same order (for example, DEFAULT (1, 'abc')).
- A PUBLIC constant should be declared at a global level.
- You can use a constant wherever you can use a literal.
- Constants are not modifiable.
- Variable declaration rules apply to constants. (See [DECLARE Variable, page 383.](#))

Example

```
PROCEDURE constants ( )
BEGIN
  DECLARE PUBLIC x CONSTANT INT DEFAULT 1234;
  DECLARE PUBLIC y CONSTANT ROW (a INT, b CHAR) DEFAULT (1, 'abc');
END
```

DECLARE CURSOR of Type Variable

You can define a new cursor variable in SQL Script by providing a unique name and optionally specifying its data type.

For details, see [Attributes of Cursors, page 356](#), [OPEN, page 399](#), [FETCH, page 392](#), and [CLOSE, page 375](#).

Syntax

```
DECLARE <variableName> CURSOR [<dataType>]
```

Remarks

- The <dataType> is optional and can be a named ROW data type or the syntax for a ROW data type.
- When declared, the cursor variable is initialized to NULL. It cannot be initialized to any other value at declaration.
- You can use the SCROLL keyword in an OPEN statement to open a cursor after a row has been fetched from a cursor, as follows:

```
DECLARE i INT;
DECLARE x CURSOR (a int) FOR SELECT COUNT(*) FROM /services/databases/system/ALL_USERS;
OPEN x SCROLL;
```

Examples

The following example returns the first name.

```
PROCEDURE p (OUT p_name VARCHAR)
BEGIN
  DECLARE c CURSOR (name VARCHAR);
  OPEN c FOR SELECT name FROM /shared/T;
  FETCH c INTO p_name;
  CLOSE c;
END
```

The following example closes and then reopens c with the same query, and later closes it and reopens it with a new query.

```
PROCEDURE p (OUT p_name VARCHAR)
BEGIN
  DECLARE c CURSOR (name VARCHAR);
  OPEN c FOR SELECT name FROM /shared/T;
  CLOSE c;
  OPEN c;
  CLOSE c;
  OPEN c FOR SELECT name FROM /share/U WHERE birthdate > '2000-01-01';
  CLOSE c;
END
```

DECLARE <cursorName> CURSOR FOR

You can define a static cursor in SQL Script by providing a unique name for it and specifying the query expression associated with the cursor.

Syntax

```
DECLARE <cursorName> CURSOR FOR <queryExpression>
```

Remarks

- The name resolution works like a standalone SELECT statement.
- Variables cannot be used in the query expression.
- Bind variables (such as '?') cannot be used.
- Declaring a static cursor is logically equivalent to preparing a statement in JDBC.
- A cursor declared in this way is like a constant: its value cannot be changed.

Examples

```
PROCEDURE p (OUT p_name VARCHAR)
BEGIN
  DECLARE c CURSOR FOR SELECT name FROM /shared/T;
  OPEN c;
  FETCH c INTO p_name;
  CLOSE c;
END
```

The procedure below returns the first name.

```
PROCEDURE p (OUT p_name VARCHAR)
BEGIN
  DECLARE c CURSOR FOR SELECT name FROM /shared/T;
  OPEN c;
  FETCH c INTO p_name;
  CLOSE c;
  ...
  --Reopen cursor
  OPEN c;
  FETCH c INTO p_name;
  CLOSE c;
END
```

The procedure below manipulates two cursors, c and d.

```
PROCEDURE p
BEGIN
  DECLARE c CURSOR (name VARCHAR);
  DECLARE d CURSOR FOR SELECT name FROM /shared/T;

  --Open a new cursor in cursor variable c
  OPEN c FOR SELECT name FROM /shared/T;

  Assign the cursor referred to by d to c
  The original cursor referred to by c is no longer accessible
  SET c = d;
```



```
--c and d cursor variables now refer to the same cursor
--Use either one to open the cursor
OPEN d; -- or OPEN c
--c.ISOPEN is true
```

The procedure below returns an opened static cursor.

```
PROCEDURE p (OUT p_cursor CURSOR (name VARCHAR))
BEGIN
    DECLARE c CURSOR FOR SELECT name FROM /shared/T;

    SET p_cursor = c;
    OPEN p_cursor;
END
```

```
--Returns an opened static cursor
PROCEDURE p (OUT p_cursor CURSOR (name VARCHAR))
BEGIN
    OPEN p_cursor FOR SELECT name FROM /shared/T;
END
```

```
PROCEDURE p (OUT p_id INTEGER, OUT p_name VARCHAR)
BEGIN
    DECLARE c CURSOR FOR SELECT id, name FROM /shared/T;
    DECLARE r ROW (id INTEGER, name VARCHAR);

    OPEN c;
    FETCH INTO c;
    CLOSE c;

    SET p_id = r.id;
    SET p_name = r.name;
END
```

```
PROCEDURE p ( )
BEGIN
    DECLARE TYPE r_type ROW (id INTEGER, name VARCHAR);
    DECLARE c CURSOR r_type;
    DECLARE r r_type;

    OPEN c FOR SELECT id, name FROM /shared/T;
    FETCH INTO c;
    CLOSE c;
END
```

DECLARE EXCEPTION

The DECLARE EXCEPTION statement in SQL Script declares an exception.

Syntax

```
DECLARE [PUBLIC] <exceptName>
EXCEPTION
```

Remarks

- An exception can be declared in a child scope that has the same name as the one declared in the parent scope. In that case, the one in the parent scope is not visible within the child scope.
- You can define exceptions by providing a unique name to each exception. See also [External Exceptions, page 371](#), [Attributes of CURRENT_EXCEPTION, page 358](#), and [Raising and Handling Exceptions, page 369](#).
- The PUBLIC keyword can only be used in the root compound statement of a PROCEDURE. It makes the exception visible outside the procedure as described in the section [External Exceptions, page 371](#). See [Compound Statements, page 365](#) for information on compound statements.

Examples

```
PROCEDURE f(IN x INTEGER)
BEGIN
  DECLARE PUBLIC illegal_arg_ex EXCEPTION;

  IF x IS NULL THEN
    RAISE illegal_arg_ex;
  END IF;
  ...
END

PROCEDURE p(IN x INTEGER, IN result BIT)
BEGIN
  CALL /shared/f(x);
  SET result = 1; -- success
  EXCEPTION
  WHEN /shared/f.illegal_arg_ex THEN
    SET result = 0; --failure
END
```

DECLARE TYPE

Defining a new data type in SQL Script is effectively a way to create an alias for a data type. The declaration can be used to make a custom string, such as aliasing FirstName to VARCHAR(24), or (more likely) for making an alias for a column set, such as aliasing ResponseCursorType to ROW(col1 VARCHAR(40), col2 INTEGER).

The data types supported in SQL Script are listed in the section [Data Types, page 348](#).

You can also declare a new data type.

Syntax

```
DECLARE [PUBLIC] TYPE <typeName> <dataType>
```

The <dataType> can be a ROW type or regular data type.

Remarks

- You can use DECLARE TYPE on CURSOR types, as in
DECLARE PUBLIC TYPE cursor_datatype_exampleA
 CURSOR (fieldA INTEGER, fieldB VARCHAR(255), fieldC DATE)
- If you alias ID to be of type INTEGER, it is a distinct type and is no longer a plain integer.
- To make the data types visible outside of a procedure, the PUBLIC keyword can only be used in the root compound statement of a procedure.

Examples

```
PROCEDURE p ( )
BEGIN
  DECLARE TYPE name_type VARCHAR(50);
  DECLARE TYPE money_type DECIMAL(18, 2);
  DECLARE TYPE id_type BIGINT;

  DECLARE a name_type DEFAULT 'Joe';
  DECLARE b money_type DEFAULT 12.34;
  DECLARE c id_type DEFAULT 1234567890;
  ...
END

PROCEDURE p ( )
BEGIN
  DECLARE TYPE r_type ROW (i INTEGER, name VARCHAR, birthdate DATE);
  DECLARE r r_type;
  DECLARE s r_type;

  SET r.id = 123;
  SET r.name = '5';
  SET r.birthdate = '1990-10-31';
  ...
END
```

DECLARE Variable

You can define a noncursor variable in SQL Script by specifying its name and data type, and initializing it with a default value. See [DECLARE CURSOR of Type Variable, page 378](#) for defining cursor variables.

Syntax

```
DECLARE <variableName>[,...] <dataType> DEFAULT <valueExpression>]
```

Remarks

- DEFAULT initializes the variable.
- You can declare more than one variable at a time, provided all the variables are of the same data type but each has a unique name.
- The <valueExpression> can use IN parameters, variables declared previously in this block, and any variables in parent blocks. In the current block, the value expression cannot use variables that are defined later. If the value expression's type does not match the variable's type, an implicit cast is performed (if possible). See [SQL Script Procedure Header, page 363](#) for information on IN parameters.
- Any variable that is not initialized with a DEFAULT clause has the value NULL.
- If the evaluation of the value expression causes an exception, declared variables that have not yet been initialized are set to NULL before entering the exception handler.

DECLARE VECTOR

DECLARE VECTOR in SQL Script declares a collection data type that is expandable, ordered, and typed. A vector requires a data type at initialization.

This section provides the general syntax for declaring a VECTOR, and describes the functionality of vectors in SQL Script. Examples are given at the end of the section.

Syntax

```
DECLARE <identifier> VECTOR (<data type>) [DEFAULT VECTOR [<value>, <value>]]
```

Base Data Types

- The DEFAULT clause is optional and can be used to initialize VECTOR values.
- A vector cannot be the base data type of another vector, so you cannot use the following declaration:

```
DECLARE vectorX VECTOR (VECTOR (CHAR));
```

- ROW is an acceptable base data type of a vector, and is necessary for any implementation of collections, as in the following example:

```
DECLARE vectorX VECTOR(ROW (a INTEGER,
b INTEGER, c CHAR, d CHAR));
```

- ROWs can also contain vectors, and a field in the ROW can be accessed through the dot notation as follows:

```
DECLARE myRow ROW(a INTEGER, v VECTOR(INTEGER));
SET myRow = ROW(1, VECTOR[9,10,11]);
SET myRow.v[2] = 9;
```

```
DECLARE vecRow VECTOR(ROW (a INTEGER, b CHAR));
SET vecRow = VECTOR[(22, 'text')];
SET vecRow[1].a = vecRow[1].a + 15;
```

Declaration

- You cannot declare a vector as a field in a CURSOR or a PIPE, so the following declaration would not be permitted:

```
DECLARE myCursor CURSOR (a VECTOR(CHAR));
```

- Vectors can be declared as PUBLIC constants or nonpublic constants. The contents of such vectors should not be modified.
- The initial contents of a CONSTANT VECTOR must be defined in a DEFAULT clause and must be literals or references to other similar type of vectors.

Assigning Values to VECTOR Elements

- An empty vector with no base type can be created by the expression VECTOR[]

- Elements in a vector can be assigned a value of NULL.

```
SET vectorX[1] = NULL;
```

- The vector is set to NULL at declaration and must be initialized before it can be used, as in the following example. Any reference to an uninitialized vector results in an error.

```
VECTOR['my text', 'your text']
```

This expression can be assigned to a compatible vector with the SET statement, as follows:

```
SET my_vector = VECTOR['my text', 'your text'];
SET your_vector = VECTOR[ROW(2,3), ROW(4,5)];
SET your_vector = my_vector;
```

In the above declaration, the contents of the source vector your_vector is copied to the target vector my_vector, and the target vector is initialized.

- Vectors can be used as parameters in procedures, and the procedures with OUT or INOUT parameters can alter the vector in the same manner as the SET statement.

```
CALL myProcedure(vectorX);
```

- After spaces are allocated in a vector by initializing the vector, elements in the vector can be accessed through square brackets, as in arrays in other programming languages. Vector indexes start at 1 and increment by 1.

```
SET vectorX[20] = 'my text';
```

```
SET yourvector[2 + index] = vectorX[20];
```

A vector index must evaluate to a numeric value. Otherwise, an error results, as in the following example:

```
SET yourvector[1 || 'text'] = 'text';
```

- If a vector index evaluates to NULL, the element reference results in NULL.
- If the target reference index is NULL, an error results, as in the following example:

```
SET vectorX[NULL] = 'text';
```

- Vectors are bound by the current allocation, but can be resized through reassignment or through system procedures.
- Vectors can be assigned to other vectors that have implicitly assignable data types. In the case where the data type is not the same, a vector is created, and all elements automatically have the CAST function run to convert the value to the target type.

Comparing Vectors

Vectors can be compared to one another if their base types are comparable. Only comparison operators such as = (equal to) and != (not equal) are supported.

Vectors are equal if they have the same number of values, and corresponding elements are equal. If either vector is NULL, the result of the comparison is unknown. If any of the elements is NULL, the result of the comparison is unknown.

Vectors and Functions

Several functions are available to modify the contents of a vector. The following functions are supported: CARDINALITY, CAST, CONCAT, EXTEND, and TRUNCATE. All vectors, regardless of their base data type, are accepted as arguments for these functions:

CARDINALITY

This function returns the number of elements allocated in the vector.

CAST

This function converts all the elements in a vector to the desired target data type. The result vector is of the same size as that of the source vector. If the vector has a NULL element, the result vector contains NULL. The source vector's data type and the target vector's data type must be compatible. For details, see the section [CAST, page 137](#).

CONCAT

This function adds two vectors that have the same data type together. If either of the vectors is NULL, an error occurs indicating that the resultant vector is NULL. Concatenating nonNULL vectors result in a new vector containing the elements from the concatenated vectors. The elements of the input vectors are added successively; that is, the elements of the first vector populates the result vector first, then the elements of the second vector populates the result vector, and so on.

Note: The || operator does the same thing as the CONCAT function.

EXTEND

This function appends the specified number of elements to a vector. The appended number of elements are assigned a NULL value, and the syntax is as follows:

```
SET vectorX = EXTEND (vectorX, 2);
```

- If the number of elements specified to be appended evaluates to NULL, this function returns NULL.
- If the vector is NULL, an error occurs, indicating that the vector is NULL.
- If the specified number is a negative number, an error occurs.

FIND_INDEX

The function searches a vector for the first occurrence of a specified value. It accepts two arguments. The first argument is any scalar value. The second argument is the vector that is searched. The index starts at 1.

- The base type of the vector and the supplied argument's data type must be comparable or implicitly castable.
- If the searched value is not found in the vector, the result is zero.
- If either the vector or the supplied argument is NULL, the result of the function is NULL.

The following example returns a value of 3:

```
DECLARE v VECTOR(INT) DEFAULT VECTOR [5, 10, 50, 100];
SET i = FIND_INDEX(50, v);
```

TRUNCATE

This function removes a specified number of elements (the “chop count”) from the end of a vector. The syntax is as follows:

```
SET vector1 = TRUNCATE (vector1, chop_count)
```

- If the chop count evaluates to NULL, this function returns NULL.
- If the chop count is negative, or exceeds the initial size of the vector, an error occurs.
- If the vector is NULL, an error occurs.
- TRUNCATE is also a TDV-supported SQL function. Refer to [TRUNCATE, page 151](#), for a description.

Examples

This section contains several examples to illustrate the functionality of vectors in SQL Script.

```
PROCEDURE vectorExampleA()
BEGIN
  DECLARE vectorX VECTOR(ROW(a int, b char));
  DECLARE vectorY VECTOR(ROW(x int, y char));

  SET vectorX = VECTOR[(11, 'one in vectorX'), (12, 'two in vectorX')];
  SET vectorY = VECTOR[(21, 'one in vectorY'), (22, 'two in vectorY')];
  CALL print(vectorX[1].b);
  CALL print(vectorX[2].b);
  IF vectorX != vectorY THEN
    CALL print(vectorY[1].y);
  END IF;
END
```

```
PROCEDURE vectorExampleB()
BEGIN
  DECLARE vectorX VECTOR(ROW(a int, b char));
  DECLARE vectorY VECTOR(ROW(x int, y char));

  SET vectorX = VECTOR[(11, 'one in vectorX'), (12, 'two in vectorX')];
  SET vectorX[1].a = vectorX[1].a + 11;
  SET vectorY = VECTOR[(5, 'one in vectorY'), (10, 'two in vectorY')];
  SET vectorX = vectorY;
  CALL PRINT(TO_CHAR(vectorX[2].a));
END
```

```
PROCEDURE vectorExampleC(OUT x VECTOR(INTEGER))
BEGIN
  DECLARE vectorX VECTOR(INTEGER);

  SET x = VECTOR[5, 55, 60];
  SET vectorX = x;
  CALL PRINT(TO_CHAR(x[1]));
END
```



```

PROCEDURE vectorExampleD()
BEGIN
DECLARE vConstM CONSTANT VECTOR(INTEGER)
DEFAULT VECTOR[1, 2];
DECLARE vConstN CONSTANT VECTOR(INTEGER)
DEFAULT VECTOR[99, vConstM[2]]
DECLARE x INTEGER;
DECLARE y INTEGER;

SET x = vConstM[1];
SET y = vConstN[1];
CALL PRINT(TO_CHAR(x));
CALL PRINT(TO_CHAR(y));
END

PROCEDURE vectorExampleE()
BEGIN
DECLARE PUBLIC vConstM CONSTANT VECTOR(INTEGER)
DEFAULT VECTOR[1, 2];
DECLARE PUBLIC vConstN CONSTANT VECTOR(INTEGER)
DEFAULT VECTOR[99, vConstM[2]];
DECLARE x INTEGER;
SET x = vConstN[2];
CALL PRINT(TO_CHAR(x));
END

PROCEDURE vectorExampleF(OUT Name VECTOR(CHAR(255)))
BEGIN
DECLARE firstName VECTOR(CHAR);
DECLARE lastName VECTOR(CHAR);

SET firstName = VECTOR['john'];
SET lastName = VECTOR['doe'];
SET Name = CONCAT(firstName, lastName);
END

PROCEDURE vectorExampleG(OUT card INTEGER)
BEGIN
DECLARE vectorX VECTOR(INTEGER);

SET vectorX = VECTOR[5, 55, 19, 15, 23];
SET card = CARDINALITY (vectorX);
END

PROCEDURE vectorExampleH(OUT ext VECTOR(INTEGER))
BEGIN
DECLARE vectorX VECTOR(INTEGER);
DECLARE NEWVECTOR VECTOR(INTEGER);

SET vectorX = VECTOR[5, 55, 19, 15, 23];
SET vectorX = EXTEND(vectorX, 2);
SET ext = vectorX;
END

PROCEDURE vectorExampleJ(OUT ext VECTOR(INTEGER))
BEGIN

```

```

DECLARE vectorX VECTOR(INTEGER);

SET vectorX = VECTOR[5, 55, 19, 15, 23];
SET vectorX = VECTOR[NULL];
SET vectorX = EXTEND(vectorX, 2);
SET ext = vectorX;
END

PROCEDURE vectorExampleK(OUT trunc VECTOR(INTEGER))
BEGIN
DECLARE vectorX VECTOR(INTEGER);
DECLARE newvector VECTOR(INTEGER);

SET vectorX = VECTOR[5, 55, 19, 15, 23];
SET newvector = TRUNCATE(vectorX, 2);
SET trunc = newvector;
END

PROCEDURE vectorExampleM(OUT trunc VECTOR(INTEGER))
BEGIN
DECLARE vectorX VECTOR(INTEGER);
DECLARE newvector VECTOR(INTEGER);

SET vectorX = VECTOR[5, 25, 30];
SET newvector = TRUNCATE(vectorX, NULL);
SET trunc = newvector;
END

```

DELETE

DELETE in SQL Script removes records from a table.

Syntax

```
DELETE FROM <table> [WHERE <conditionalExpression>]
```

Remarks

- Any legal DELETE statement that the system accepts can be used as a standalone SQL Script statement.
- Variables are allowed in a SQL statement anywhere literals are allowed.

Examples

```

PROCEDURE p ( )
BEGIN
DELETE FROM /shared/scores;
INSERT INTO /shared/scores VALUES ('Joe', 1001);
UPDATE /shared/.scores SET score=1239 WHERE name='Sue';
END

PROCEDURE p (IN p_name VARCHAR, IN new_score)

```

```

BEGIN
DELETE FROM /shared/scores WHERE name=p_name;
INSERT INTO /shared/scores VALUES (p_name, new_score);
UPDATE /shared/.scores SET score=new_score WHERE name=p_name;
END

PROCEDURE p (IN y VARCHAR)
BEGIN
--T has columns x and y
--The following y refers to the column, not the parameter
DELETE FROM /shared/T WHERE x = y;
END

```

DROP TABLE

Removes a table definition and all the data, indexes, triggers, constraints and permission specifications for that table.

Syntax

```
DROP TABLE [IF EXISTS] table_name;
```

Remarks

- DROP TABLE throws an error if the table does not exist, or if other database objects depend on it.
- DROP TABLE IF EXISTS does not throw an error if the table does not exist. It throws an error if other database objects depend on the table.

DROP INDEX

Deletes the index in a table.

Syntax

```
DROP INDEX index_name ON table_name;
```

EXECUTE IMMEDIATE

The EXECUTE IMMEDIATE statement in SQL Script dynamically executes certain SQL statements.

Syntax

```
EXECUTE IMMEDIATE <valueExpr>
```

Remarks

- The <valueExpr> must evaluate to a string type (CHAR or VARCHAR). The text in this string is executed as SQL.
- This form of dynamic SQL is useful mainly for INSERT, UPDATE, and DELETE statements. It has no value to SELECT, because the selections cannot be assigned to anything. See the OPEN FOR statement used in [OPEN](#), [page 399](#) for information about how to perform a dynamic SELECT.

Example

```
PROCEDURE drop (IN table_name VARCHAR)
BEGIN
  DECLARE sql_stmt VARCHAR;

  SET sql_stmt
    = CAST(CONCAT('DELETE FROM ', table_name) AS VARCHAR);
  EXECUTE IMMEDIATE sql_stmt;
END
```

FIND_INDEX

Returns the index of the first object in an array. Return zero if nothing is found. If the first item in the array matches the first argument, then 1 is returned.

Syntax

```
<array>.find_index{<varList>}
```

Example

```
PROCEDURE ss1(out i int)
BEGIN
  declare v vector(int) default vector [1,2,3,4];
  set i = find_index(-5, v);
END
```

FETCH

The FETCH statement is used in SQL Script to read one row from an open cursor.

Syntax

```
FETCH <cursor> INTO <varList>
```

The variable list can be a list of variables (same number as the number of projections) or a ROW variable with the right schema. For information on ROW, see [DECLARE CURSOR of Type Variable, page 378](#).

Remarks

- The <varList> works like the SELECT INTO clause. (See [SELECT INTO, page 403](#).)
- It is illegal to fetch from a cursor that is not open.
- Fetching past the last row does not cause an error. The variables are not altered and the FOUND attribute is set to FALSE. See [Attributes of Cursors, page 356](#) for details.
- You can specify the direction of the fetch to be NEXT or FIRST. These words must be used along with the keyword FROM, as follows:

```
FETCH NEXT FROM x INTO i;  
FETCH FIRST FROM x INTO i;
```

If no fetch orientation is specified, NEXT is the default.

If the orientation is NEXT, the fetch behaves as it always has: it fetches the current row’s data into the target variables.

If FIRST is specified as the orientation, the cursor must be a SCROLL cursor, otherwise an error results. See [DECLARE CURSOR of Type Variable, page 378](#).

If the orientation specified is FIRST, the cursor is repositioned to the first row, and the first row’s data is placed in the target variables.

Errors

The following table describes the errors that can occur while executing a FETCH statement.

| Error Message | Cause |
|----------------------|--|
| Uninitialized cursor | The cursor variable is used, but is not initialized at the time it is fetched. |
| Cursor is not open | Cursor was closed when the fetch was attempted. |

FOR

FOR statements are used in SQL Script to loop through a query or cursor. FOR statements have two formats.

Syntax1

Used to loop across a query expression.

```
[<label>:]
FOR <loopVariable> AS [<cursorName> CURSOR FOR]
<queryExpression> DO
<statements>
END FOR [<label>]
```

Syntax2

Used to loop across a cursor. For details, see [DECLARE CURSOR of Type Variable, page 378](#).

```
[<label>:]
FOR <loopVariable> AS <cursorVariable> DO
<statements>
END FOR [<label>]
```

Remarks

- The <label> is an optional identifier to name the block. This is for use with the LEAVE and ITERATE statements. See [LEAVE, page 397](#) and [ITERATE, page 397](#).
- If a beginning label is present, the end label is not required. If no beginning label is present, it is illegal to have an end label. If both the beginning and end labels are present, both must have the same identifier.
- There can be zero or more statements in the <statements> area.
- The FOR statement declares the loop variable to be of the proper type to match the query expression (a ROW). You do not have to declare that variable elsewhere. The variable is only legal within the loop block. This variable can have the same name as another variable in the current scope (or a parent scope), but it cannot have the same name as a parameter to the procedure. If it does have the same name, the same rules apply as for declaring variables in a compound statement. See [Compound Statements, page 365](#) for details.
- If a cursor variable is provided in the first format (Syntax 1), it is also declared at this point. You do not declare it separately. This variable is set to be a cursor for the provided query expression.
- The cursor is opened when it starts. You do not have to open the cursor. It then fetches rows (use FETCH) one at a time and assigns the row into the loop

variable. This makes it possible to operate on each row one at a time. The cursor is closed automatically when the loop ends. See [FETCH](#), page 392.

If you open the cursor (and even fetch a few rows), the FOR loop picks up where the cursor is. If you do not open the cursor, the FOR statement opens it for you.

The FOR loop closes the cursor no matter how the loop exits (even with a LEAVE statement).

- When a FOR loop is passed a cursor, it opens the cursor if it is not already open.
- After the FOR loop, the cursor is closed. Even if you try to LEAVE the FOR loop, the cursor is closed. If you try to close a cursor that was used by a FOR loop, an error occurs.

Example

```
--Returns the average of all scores
PROCEDURE avr_score(OUT result INTEGER)
BEGIN
  DECLARE crs CURSOR FOR
    SELECT name, score FROM /shared/U ORDER BY score DESC;
  DECLARE total INTEGER DEFAULT 0;
  DECLARE cnt INTEGER DEFAULT 0;

  OPEN crs;
  FOR r AS crs DO
    SET total = total + r.score;
    SET cnt = cnt + 1;
  END FOR;
  SET result = total/cnt;
END
```

IF

The IF statement is used in SQL Script to evaluate a condition.

Syntax

```
IF <conditionalExpression> THEN
  <statements>
[ELSEIF
  <statements> ...]
[ELSE <statements>]
END IF
```

The <statements> area contains a sequence of zero or more statements. Each statement is followed by a semicolon.

Example

```

PROCEDURE "max" (IN a INTEGER, IN b INTEGER, OUT "max" INTEGER)
BEGIN
  IF a IS NULL OR b IS NULL THEN
    SET "max" = NULL;
  ELSEIF a > b THEN
    SET "max" = b;
  ELSEIF b > a THEN
    SET "max" = b;
  ELSE
    SET "max" = a;
  END IF;
END

```

INSERT

The INSERT INTO statement is used in SQL Script to insert values into the columns of a table. Almost any INSERT statement can be used as a standalone SQL Script statement.

Variables are allowed in a SQL statement anywhere literals are allowed.

Syntax

```

INSERT INTO table_name[(column_A,column_X,...)]
VALUES ('value1','value X',...);

```

Remarks

- Specification of the column names is optional. The VALUES list contains comma-separated values for insertion into the specified columns.
- The INSERT INTO statement can also be used to insert a complete row of values without specifying the column names. Values must be specified for every column in the table in the order specified by the DDL. If the number of values is not the same as the number of columns in the table, or if a value is not allowed for a particular data type, an exception is thrown.
- The syntax of INSERT is extended to allow PIPE variables to be used where a table name is normally used. This is how rows are inserted into a PIPE. See [PIPE Modifier, page 364](#).

Examples

```

PROCEDURE p1 (OUT result PIPE(C1 VARCHAR(256)))
BEGIN
  INSERT INTO result(C1) VALUES(some_variable);
END

PROCEDURE p2 ( )
BEGIN

```



```
INSERT INTO birthdays(person_name,"birth date",'annotation') VALUES('Chris Smith','2006-12-20','Last years
gift: Watch');
END
```

ITERATE

The ITERATE statement is used in SQL Script to continue the execution of the specified label.

Syntax

```
ITERATE <label>
```

Remark

The ITERATE statement is equivalent to continue in Java. It jumps to the end of the loop block and causes the loop to evaluate its condition (if available) and loop back to the top.

Example

```
PROCEDURE
BEGIN
  DECLARE c CHAR(1);
  DECLARE ix INTEGER DEFAULT 1;
  SET result = '';

  label a:
  WHILE ix <= LENGTH(s) DO
    SET c = CAST(SUBSTRING(s, ix, 1) AS CHAR(1));
    SET ix = ix + 1;
    IF c = '' THEN
      ITERATE label_a;
    END IF;
    SET result = CAST(CONCAT(result, c) AS VARCHAR);
  END WHILE;
END
```

LEAVE

The LEAVE statement is used in SQL Script to abort execution of the current block.

Syntax

```
LEAVE <label>
```

Remark

The LEAVE statement is equivalent to using break in Java. It aborts the current loop or compound statement block, without throwing an error.

Example

```
--Pads s with padChar so that s has at least width length.
PROCEDURE padr (IN s VARCHAR, IN width INTEGER, IN padChar VARCHAR, OUT result VARCHAR)
L-padr:
BEGIN
  --Returns null if any parameter is null
  IF s IS NULL OR width IS NULL OR padChar IS NULL THEN
    LEAVE L-padr;
  END IF;

  ...
END
```

LOOP

The LOOP statement is used in SQL Script for looping through the current block.

Syntax

```
[<label>:] LOOP
<statements>
END LOOP [<label>]
```

This sample statement loops forever. You need to use a LEAVE statement to exit it.

Remarks

- The label is an optional identifier to name the block. This is for use with the LEAVE and ITERATE statements. See [LEAVE, page 397](#) and [ITERATE, page 397](#).
- If a beginning label is present, the end label is not required. If no beginning label is present, then it is illegal to have an end label. If both the beginning and end labels are present, then both must have the same identifier.
- There can be zero or more statements in the <statements> area.

Example

```
This example pads s with padChar so that s has at least width length.
PROCEDURE padr(IN a VARCHAR, IN width INTEGER, IN padChar VARCHAR, OUT result VARCHAR)
--pad result with padChar
SET result = s;
```

```

L-loop:
LOOP
IF LENGTH(result) >= width THEN
  LEAVE L_loop;
END IF;
SET result = CAST(CONCAT(result, padChar) AS VARCHAR);
END LOOP;
END

```

OPEN

The OPEN statement is used in SQL Script to open a cursor. Two types of OPEN statements are available, one to open a static cursor and another to open a variable cursor. The OPEN statement for a variable cursor can specify whether it is for a query expression or a value expression. See [Value Expressions, page 352](#).

Syntax (Open Static Cursor)

```
OPEN <cursor>
```

Syntax (Open Variable Cursor)

```
OPEN <cursorVariableName> FOR <queryExpression>
```

Remarks

- A cursor variable can be opened and initialized using a dynamic SQL statement as follows:

```
OPEN <cursorVariableName> FOR <valueExpression>
```

- OPEN is similar to preparing a statement for execution.
- Run-time errors, such as insufficient privileges, are not caught until a statement is executed.
- The syntax for the open static cursor statement works on both static and variable cursors, although you get an error if you open an uninitialized cursor variable.
- It is illegal to open a cursor that is already open.

Errors

Standard parser and resolver errors can result from the SELECT statement in the FOR clause. The following table describes the errors that can occur when executing an OPEN statement.

| Error Message | Cause |
|----------------------|--|
| Cannot open a PIPE | An attempt is made to open a PIPE variable. |
| Uninitialized cursor | A cursor variable is used and is not initialized at the time it is opened. |
| Cursor already open | OPEN was invoked when the cursor was already open. |

PATH

You can define paths to resources in SQL Script by providing a unique names to each path. PATH is similar to IMPORT in Java.

Remarks

- PATH should be specified in the first BEGIN/END as the first statement after BEGIN.
- Wherever you can use a variable, you can use PATH.
- PATH can be used to fully qualify unqualified tables or procedures used in the FROM clause, and CALL and INSERT/DELETE/UPDATE statements.

Syntax

PATH <full path>

Example

```
PROCEDURE p_path1(out outgoing int)
BEGIN
    PATH /users/composite/test/views;
    DECLARE public x constant int default 0;
    DECLARE public y constant int default 5;
    DECLARE public z constant int default 0;
    DECLARE public e1 exception;
    SET outgoing = y;
EXCEPTION
    WHEN /users/composite/test/views/p_path1.e1 THEN
END
```

RAISE

The RAISE statement is used in SQL Script to raise an exception.

Syntax

```
RAISE [<exceptionName>] [VALUE [<valueExpression>]]
```

Remarks

- The value expression must resolve to a string. (See [Value Expressions, page 352](#).)
- The <exceptionName> can be any exception that is defined in the current scope, a parent scope, or that has a qualified name (such as a system exception).
- A name is required if this statement is outside of an exception handler. When inside an exception handler and when no name is used, the current exception is re-raised.
- The <valueExpression> can optionally be set on an exception. If not present, the value defaults to NULL. The value be implicitly cast (if necessary) to be assigned into the exception.

You can change the value of an exception when re-raising it by including the VALUE clause but no exception name.

Examples

```
PROCEDURE square (IN x INTEGER)
BEGIN
  DECLARE illegal_parameter_ex EXCEPTION;

  IF x IS NULL THEN
    RAISE illegal_parameter_ex;
  END IF;

  ...
END

PROCEDURE p (IN x INTEGER)
BEGIN
  DECLARE illegal_parameter_ex EXCEPTION;

  IF x < 0 THEN
    RAISE illegal_parameter_ex VALUE 'x must be > 0. x='||x;
  END IF;

  ...
END
```

REPEAT

The REPEAT statement is used in SQL Script to repeat specific statements under specific conditions.

Syntax

```
[<label>:] REPEAT
<statements>
UNTIL <conditionalExpression>
END REPEAT [<label>]
```

Remarks

- The label is an optional identifier to name the block. The REPEAT statement is for use with the LEAVE and ITERATE statements. See [LEAVE, page 397](#) and [ITERATE, page 397](#).
- If a beginning label is present, the end label is not required. If no beginning label is present, it is illegal to have an end label. If both the beginning and end labels are present, both must have the same identifier.
- The <statements> area can have zero or more statements.

Example

```
--Returns the root of ID
PROCEDURE
BEGIN
  DECLARE parent_ID INTEGER DEFAULT ID;
  REPEAT
    SET result = parent_ID;
    CALL /shared/parent_of (result, parent_ID);
  UNTIL parent_ID IS NULL
  END REPEAT;
END
```

ROLLBACK

If you are inside a compound statement with an independent transaction, you can invoke ROLLBACK in SQL Script to roll back the transaction. See [Compound Statements, page 365](#).

Syntax

```
ROLLBACK
```

Remark

It is illegal to call ROLLBACK in a compound statement that is not declared INDEPENDENT.

Example

```
PROCEDURE p ( )
BEGIN INDEPENDENT TRANSACTION
  INSERT INTO /shared/T (name, score) VALUES ('Joe', 123);
  ROLLBACK;
END
```

SELECT INTO

Any SELECT statement that the system accepts can be used in SQL Script as a standalone SQL Script statement, as long as it uses the SELECT INTO format.

Syntax

```
SELECT <projections> INTO <varListOrRowVariable>
FROM . . .
```

Remarks

- A standalone SELECT statement without the INTO clause is disallowed and discarded by the optimizer because it would do nothing to the program state.
- Variables are allowed in a SQL statement anywhere a literal of the same type is allowed.
- The BOOLEAN and ROW types are not supported in SQL.
- There is no special syntax for noting that something is a variable instead of a column in SQL statements, so be cautious when declaring a variable's name. If there is a conflict, the name is interpreted as a column name and not a variable name.
- When using SELECT INTO, the cursor must return a single row. If it returns no rows or multiple rows, an exception is raised.
- Use of SELECT INTO is sometimes called an "implicit cursor" because it is opened, fetches one row, and is closed in one statement.

Example

```
PROCEDURE selinto_ex ( )
BEGIN
  DECLARE a INTEGER;
  DECLARE b DATE;
```

```
SELECT col1, col2 INTO a, b FROM T WHERE x = 1;
END
```

SET

The SET statement in SQL Script is an assignment statement that assigns a value to a variable.

Syntax

```
SET <varName> = <value>
```

Remarks

- Values are coerced (implicitly cast) if that is possible.
- ROW values can be assigned to ROW variables only if each of the fields in the ROW variable could be assigned independently. Fields are coerced (implicitly cast) as required.
- A cursor variable with a type can be assigned from any cursor with the same ROW type, or to any cursor variable with exactly the same ROW type.
- A cursor variable without a type can be assigned from any cursor, or to any cursor. Assigning to a typed cursor forces a runtime schema match comparison and raises an exception on a mismatch.
- Assigning a cursor creates a reference to the original cursor’s state. This means that opening, closing, or fetching from the original cursor or the variable has the same effect and alters what the other would see. See [OPEN](#), [page 399](#), [CLOSE](#), [page 375](#), and [FETCH](#), [page 392](#) for details on opening, closing, and fetching actions on cursors.

Errors

The following table describes the errors that can occur when executing a SET statement.

| Error Message | Cause |
|---|--|
| Cannot alter the value of an IN parameter | The specified variable is an IN parameter. |

TOP

A TOP clause in a SELECT statement specifies the number of records to return, starting with the first record in the table.

Syntax

```
SELECT TOP <number> <column_name>
FROM <table>
```

Remarks

- TOP can improve performance by limiting the number of records returned, especially when very large tables are involved.
- The number argument is an integer representing how many rows to return.
- Use TOP with the ORDER BY clause to make sure your specified number of rows is in a defined order.

Example

```
PROCEDURE LookupProduct(OUT result CURSOR(ProductDescription VARCHAR(255)))
BEGIN
  OPEN result FOR SELECT
    TOP 5 products.ProductDescription
  FROM /shared/examples/ds_inventory/tutorial/products products;
END
```

UPDATE

An UPDATE statement in SQL Script updates records in a table.

Syntax

```
UPDATE <table>
SET <column> = <valueExpression> [, <column> = <valueExpression>]*
[WHERE <conditionalExpression>]
```

Remarks

- Any UPDATE statement that the system accepts can be used as a standalone SQL Script statement.
- Variables are allowed in a SQL statement anywhere a literal is allowed.
- The WHERE clause is optional. The rules for the WHERE clause of an UPDATE statement is the same as the rules for WHERE clause of a SELECT statement.
- The following subqueries in the SET clause are not allowed:

```
UPDATE <table1> SET x = (SELECT y FROM <table2>)
```

Examples

```
PROCEDURE p ( )
BEGIN
```

```

DELETE FROM /shared/scores;
INSERT INTO /shared/scores VALUES ('Joe', 1001);
UPDATE /shared/.scores SET score=1239 WHERE name='Sue';
END

PROCEDURE p (IN p_name VARCHAR, IN new_score)
BEGIN
  DELETE FROM /shared/scores WHERE name=p_name;
  INSERT INTO /shared/scores VALUES (p_name, new_score);
  UPDATE /shared/.scores SET score=new_score WHERE name=p_name;
END

```

WHILE

The WHILE statement is used in SQL Script to execute certain statements as long as specific conditions are met.

Syntax

```

[<label>:] WHILE <conditionalExpression> DO
  <statements>
END WHILE [<label>]

```

Remarks

- The <label> is an optional identifier to name the block.
- The WHILE statement is for use with the LEAVE and ITERATE statements. See [LEAVE, page 397](#) and [ITERATE, page 397](#).
- If a beginning label is present, the end label is not required. If no beginning label is present, it is illegal to have an end label. If both the beginning and end labels are present, both must have the same identifier.
- The <statements> area can have zero or more statements.

SQL Script Examples

This section contains several examples illustrating the use of the SQL Script language. All the examples assume a user named test in the domain composite.

- [Example 1 \(Fetch All Rows\), page 407](#)
- [Example 2 \(Fetch All Categories\), page 407](#)
- [Example 3 \(User-Defined Type\), page 408](#)
- [Example 4 \(User-Defined Type\), page 408](#)
- [Example 5 \(Pipe Variable\), page 408](#)

- [Example 6 \(Dynamic SQL Extract with Individual Inserts\)](#), page 408
- [Example 7 \(Dynamic SQL Inserts by Variable Name\)](#), page 409
- [Example 8 \(Prepackaged Query\)](#), page 409
- [Example 9 \(Exception Handling\)](#), page 409
- [Example 10 \(Row Declaration\)](#), page 410
- [Example 11 \(Avoiding Division-by-Zero Errors\)](#), page 410

Example 1 (Fetch All Rows)

This script iterates through a table and fetches all the rows. It assumes a Northwind access database named access and gathers all the categories in the table Categories.

```
PROCEDURE fetchExample1 (OUT category CHAR)
BEGIN
    DECLARE temp CHAR;
    DECLARE f CURSOR FOR SELECT Categories.CategoryName
        FROM /shared/access/Categories Categories;

    SET category = "";
    OPEN f;
    FETCH f INTO temp;
    -- Must call FETCH first, otherwise FOUND is false.
    WHILELOOP:
    WHILE f.FOUND
    DO
        BEGIN
            SET category = CAST(CONCAT(CONCAT(category, ','), temp)AS CHAR(255));
            FETCH f INTO temp;
        END;
    END WHILE;
    CLOSE f;
END
```

Example 2 (Fetch All Categories)

This example is similar to [Example 1 \(Fetch All Rows\)](#), page 407, but it fetches all the categories.

```
PROCEDURE fetchExample2 (OUT category CHAR)
BEGIN
    DECLARE temp CHAR DEFAULT "";

    SET category = "";
    FOR x AS SELECT Categories.CategoryName
        FROM /shared/access/Categories Categories
    DO
        SET temp = x.categoryName;
```

```

        SET category = CAST(CONCAT(CONCAT(category, ' '), temp) AS CHAR);
    END FOR;
END

```

Example 3 (User-Defined Type)

This example declares a user-defined type named `udt`, and uses it in another user-defined type `b`.

```

PROCEDURE type_example1 ()
BEGIN
    DECLARE PUBLIC TYPE udt INTEGER;
    DECLARE TYPE b ROW (a INTEGER, b udt, c VARCHAR(255));
END

```

Example 4 (User-Defined Type)

```

PROCEDURE type_example2 ()
BEGIN
    -- b is defined in Example 3 \(User-Defined Type\), page 408
    DECLARE test /shared/type_example1.b;

    SET test.a = 123;
    SET test.b = 345;
    SET test.c = 'hello';
END

```

Example 5 (Pipe Variable)

This example inserts the categories from the Northwind database into a PIPE variable.

```

PROCEDURE pipe_example2 (OUT param1 PIPE (col1 CHAR), IN param2 INT)
BEGIN
    FOR x AS SELECT Categories.CategoryName, Categories.CategoryId
        FROM /shared/access/Categories Categories
    DO
        IF x.CategoryId = param2 THEN
            INSERT INTO param1 (col1) VALUES (x.categoryName);
        END IF;
    END FOR;
    CLOSE param1;
END

```

Example 6 (Dynamic SQL Extract with Individual Inserts)

This example extracts data from a SELECT statement and uses an INSERT statement with the data. It extract the values and insert the values one by one.

```

PROCEDURE dynamic_sql_example ()
BEGIN
    DECLARE sqltext VARCHAR DEFAULT
        'INSERT INTO /shared/updates(c_varchar) VALUES('';

```

```

DECLARE temp VARCHAR;

FOR x AS SELECT Categories.CategoryName
      FROM /shared/access/Categories Categories
DO
  SET temp = CAST(sqltext || x.categoryName || '') AS VARCHAR);
  EXECUTE IMMEDIATE temp;
END FOR;
END

```

Example 7 (Dynamic SQL Inserts by Variable Name)

This example creates a dynamic SQL string to insert data from a variable. Instead of extracting the values, it calls the value by variable name.

```

PROCEDURE dynamic_sql_example2 ()
BEGIN
  DECLARE sql2 VARCHAR DEFAULT
    'INSERT INTO /shared/updates(e_varchar) VALUES(';
  DECLARE temp CHAR;

  FORLOOP:
  FOR x AS SELECT Categories.CategoryName
        FROM /shared/access/Categories Categories
  DO
    SET temp = CAST(sql2 || 'x.categoryName)' AS CHAR);
    EXECUTE IMMEDIATE temp;
  END FOR;
END

```

Example 8 (Prepackaged Query)

This example calls a prepackaged query, and returns the first row of data. It assumes that the user has a prepackaged query named, `pqAccess`, under the `shared` folder.

```

PROCEDURE prepackaged_query_example ()
BEGIN
  -- Declare a cursor to retrieve from the prepackaged query
  DECLARE myRow ROW(a1 INT, a2 VARCHAR, a3 VARCHAR, a4 DECIMAL, a5 INT,          a6
DECIMAL, a7 VARCHAR, a8 VARCHAR);
  DECLARE crs cursor(a1 int, a2 VARCHAR, a3 VARCHAR, a4 DECIMAL, a5 INT,          a6
DECIMAL, a7 VARCHAR, a8 VARCHAR);

  CALL /shared/pqAccess(crs);
  -- Fetch the first row
  FETCH crs INTO myRow;
END

```

Example 9 (Exception Handling)

This example shows how to raise `EXCEPTION`.

```

PROCEDURE exception_example (OUT has_error INT)
BEGIN
    DECLARE too_many_categories EXCEPTION;
    DECLARE no_categories EXCEPTION;
    DECLARE category_count INT DEFAULT 0;

    SELECT COUNT(Categories.CategoryName) INTO category_count
    FROM /shared/access/Categories Categories;
    IF category_count > 5 THEN
        RAISE too_many_categories;
    ELSEIF category_count = 0 THEN
        RAISE no_categories;
    END IF;
    SET has_error = 0;
EXCEPTION
    WHEN too_many_categories OR no_categories THEN
        SET has_error = 1;
END

```

Example 10 (Row Declaration)

This example shows how to declare **ROW**.

```

PROCEDURE row_example()
BEGIN
    DECLARE category_row ROW (categoryid INT, category CHAR);
    DECLARE f CURSOR FOR SELECT Categories.CategoryId, Categories.CategoryName
    FROM /shared/access/Categories Categories;

    OPEN f;
    FETCH f INTO category_row;
    CLOSE f;
END

```

Example 11 (Avoiding Division-by-Zero Errors)

This example prevents “divide by zero” errors.

```

PROCEDURE divide
(IN dividend INT, IN divisor INT, OUT result INT, OUT message CHAR)
BEGIN
    DECLARE divide_by_zero EXCEPTION;

    IF divisor = 0 THEN
        RAISE divide_by_zero value 'Divided by zero error';
    END IF;
    SET result = dividend/divisor;
EXCEPTION
    WHEN divide_by_zero THEN
        SET message = CURRENT_EXCEPTION.MESSAGE;
END

```

TDV Built-in Functions for XQuery

TDV offers built-in XQuery extension functions that users can add within the text of XQuery procedures. They are meant to assist in writing and executing SQL statements from within XQuery.

This topic describes these XQuery extension functions:

- [executeStatement](#), page 411
- [formatBooleanSequence](#), page 412
- [formatDateSequence](#), page 412
- [formatDecimalSequence](#), page 413
- [formatDoubleSequence](#), page 413
- [formatFloatSequence](#), page 413
- [formatIntegerSequence](#), page 414
- [formatStringSequence](#), page 414
- [formatTimeSequence](#), page 415
- [formatTimestampSequence](#), page 415

executeStatement

This function executes the given SQL statement.

Syntax

composite:executeStatement (\$statement as item(), \$arguments as node(*)*)

Example

```
declare variable $values := <a><b>1</b><b>3</b></a>;
composite:executeStatement ('SELECT * FROM /shared/examples/ds_inventory/products WHERE ProductID > {0}
AND ProductID < {1}', $values//b)
```

Result

The output is of the form document():

```
<results>
<result>
  <ProductID>2</ProductID>
  <ProductName>Mega Zip 750MB USB 2.0</ProductName>
```

```

    <ProductDescription>Mega Zip 750 MB</ProductDescription>
    <CategoryID>1</CategoryID>
    <SerialNumber>5-76-9876</SerialNumber>
    <UnitPrice>187.67</UnitPrice>
    <ReorderLevel>5</ReorderLevel>
    <LeadTime>7 Days</LeadTime>
  </result>
</results>

```

formatBooleanSequence

This function formats a sequence of booleans as a comma-separated list of SQL literals.

Syntax

composite:formatBooleanSequence (\$values as node(*)*)

Example

```

declare variable $values := <a><b>0</b><b>1</b></a>;
<result>{composite:formatBooleanSequence ($values//b)}</result>

```

Result

The output is of the form xs:string:

```

<result>false,true</result>

```

formatDateSequence

This function formats a sequence of dates as a comma-separated list of SQL literals.

Syntax

composite:formatDateSequence (\$values as node(*)*)

Example

```

declare variable $values := <a><b>2012-06-01</b><b>2012-07-01</b></a>;
<result>{composite:formatDateSequence ($values//b)}</result>

```

Result

The output is of the form xs:string:

```

<result>'2012-06-01','2012-07-01'</result>

```


formatDecimalSequence

This function formats a sequence of decimals as a comma-separated list of SQL literals.

Syntax

composite:formatDecimalSequence (\$values as node(*)*)

Example

```
declare variable $values := <a><b>1.0</b><b>2.0</b></a>;
<result>{composite:formatDecimalSequence ($values//b)}</result>
```

Result

The output is of the form xs:string:

```
<result>1.00,2.00</result>
```

formatDoubleSequence

This function formats a sequence of doubles as a comma-separated list of SQL literals.

Syntax

composite:formatDoubleSequence (\$values as node(*)*)

Example

```
declare variable $values := <a><b>1.0</b><b>2.0</b></a>;
<result>{composite:formatDoubleSequence ($values//b)}</result>
```

Result

The output is of the form xs:string:

```
<result>1.0,2.0</result>
```

formatFloatSequence

This function formats a sequence of floats as a comma-separated list of SQL literals.

Syntax

composite:formatFloatSequence (\$values as node()*)

Example

```
declare variable $values := <a><b>1</b><b>2</b></a>;
<result>{composite:formatFloatSequence ($values//b)}</result>
```

Result

The output is of the form xs:string:

```
<result>1.0,2.0</result>
```

formatIntegerSequence

This function formats a sequence of integers as a comma-separated list of SQL literals.

Syntax

composite:formatIntegerSequence (\$values as node()*)

Example

```
declare variable $values := <a><b>1</b><b>2</b></a>;
<result>{composite:formatIntegerSequence ($values//b)}</result>
```

Result

The output is of the form xs:string:

```
<result>1,2</result>
```

formatStringSequence

This function formats a sequence of strings as a comma-separated list of SQL literals.

Syntax

composite:formatStringSequence (\$values as node()*)

Example

```
declare variable $values := <a><b>1</b><b>2</b></a>;
<result>{composite:formatStringSequence ($values//b)}</result>
```

Result

The output is of the form xs:string:

```
<result>'1',2'</result>
```

formatTimeSequence

This function formats a sequence of times as a comma-separated list of SQL literals.

Syntax

```
composite:formatTimeSequence ($values as node()*)
```

Example

```
declare variable $values := <a><b>00:00:00</b><b>23:59:59</b></a>;
<result>{composite:formatTimeSequence ($values//b)}</result>
```

Result

The output is of the form xs:string:

```
<result>'00:00:00','23:59:59'</result>
```

formatTimestampSequence

This function formats a sequence of timestamps as a comma-separated list of SQL literals.

Syntax

```
composite:formatTimestampSequence ($values as node()*)
```

Example

```
declare variable $values := <a><b>2012-01-01 00:00:00</b><b>2012-12-31 23:59:59</b></a>;
<result>{composite:formatTimestampSequence ($values//b)}</result>
```

Result

The output is of the form xs:string:

```
<result>'2012-01-01 00:00:00','2012-12-31 23:59:59'</result>
```


Java APIs for Custom Procedures

Procedures are used to generate or act on data, much like a SELECT or an UPDATE statement. The custom Java APIs are provided with the build at this location:

`<TDV_install_dir>\apps\extension\docs\com\compositesw\extension`

This topic describes TDV's extended Java APIs that support custom procedures in the system.

- [com.compositesw.extension, page 417](#)
- [CustomCursor, page 418](#)
- [CustomProcedure, page 420](#)
- [CustomProcedureException, page 423](#)
- [ExecutionEnvironment, page 424](#)
- [ParameterInfo, page 428](#)
- [ProcedureConstants, page 432](#)
- [ProcedureReference, page 435](#)

com.compositesw.extension

The extension package provides a mechanism for you to write custom procedures. All interfaces for custom Java procedures are available in this package.

`com.compositesw.extension`

Interface Summary

| | |
|--|--|
| CustomCursor, page 418 | Defines a cursor type. |
| CustomProcedure, page 420 | Defines a custom procedure. |
| ExecutionEnvironment, page 424 | Used by a procedure to interact with the TDV Server. |
| ProcedureConstants, page 432 | Contains constants used in the interfaces of the <code>com.compositesw.extension</code> package. |

Interface Summary

| | |
|--|---|
| ProcedureReference , page 435 | Provides a way to invoke a procedure and fetch its output values. |
|--|---|

Class Summary

| | |
|---|--|
| ParameterInfo , page 428 | Contains information about a custom procedure’s input or output parameter. |
|---|--|

Exception Summary

| | |
|--|---|
| CustomProcedureException , page 423 | Exception thrown by the methods of the extension APIs in the package com.compositesw.extension. |
|--|---|

CustomCursor

The CustomCursor interface returns a cursor type. All custom cursors must implement this interface.

```
public interface CustomCursor
```

A custom procedure with just one output cursor can implement both the [CustomProcedure](#), page 420 and the CustomCursor interfaces to avoid needing another class. A custom procedure with more than one output cursor should use inner classes or separate classes.

Class Summary

| | |
|--|--|
| ExecutionEnvironment , page 424 | Lets a procedure interact with the TDV Server. |
|--|--|

Method Summary

| | |
|-----------------|--|
| void | close , page 419 Frees the resources. |
| ParameterInfo[] | getColumnInfo , page 419 Returns the metadata for the cursor. |
| Object[] | next , page 419 Returns the next row, or NULL when done. |

Method Detail

close

```
public void close()
```

This method is called to free resources. Calling this method multiple times has no effect, and no exception is thrown.

Throws

This method throws [CustomProcedureException](#), page 423.

getColumnInfo

```
public ParameterInfo[] getColumnInfo()
```

This method is called to get the metadata for the custom cursor.

Returns

This method returns the metadata for the cursor. A NULL value might be returned to indicate that the caller should retrieve the metadata information by calling [ProcedureReference.getParameterInfo](#), page 439.

Throws

This method throws [CustomProcedureException](#), page 423 if the cursor has been closed. This method throws [CustomProcedureException](#), page 423 or [SQLException](#) if an error occurs while fetching the metadata.

next

```
public Object[] next()
```

This method is called when more metadata is needed.

Returns

This method returns the next row, or NULL when done.

Throws

This method throws [CustomProcedureException, page 423](#) if the cursor has been closed. This method throws [CustomProcedureException, page 423](#) or `SQLException` if an error occurs while fetching the metadata.

CustomProcedure

The CustomProcedure interface defines a custom procedure. Any class implementing this interface should define an empty constructor so that the custom procedure can be properly instantiated.

```
public interface CustomProcedure
```

This interface extends [ProcedureReference, page 435](#).

All methods in the CustomProcedure except for the constructor can throw a [CustomProcedureException, page 423](#) if they encounter an error condition. Any exception thrown from these methods (including runtime exceptions) causes an error on the current action to be passed up as a system error.

| Method Summary | |
|----------------|--|
| void | commit, page 421 |
| String | getDescription, page 421 |
| String | getName, page 422 |
| void | initialize, page 422 |
| void | rollback, page 428 |

Serialization

The custom procedure class can implement the `java.lang.Serializable` interface to carry the compensation state across a server restart. Variables that do not need to be restored after a restart should be marked as transient.

Life Cycle

The life cycle of a custom procedure object is defined as follows:

- Introspection time—A constructor is used to make an object, introspection methods are used to read method signatures, and then the object is discarded.

- Runtime setup—A constructor is used to make a new object and [initialize, page 422](#) is called.
- Runtime execution—Call [invoke, page 439](#) first, then retrieve and read from output parameter values, and then retrieve output values. You can do setup and then not invoke at all.
- Runtime closing—If the object was invoked, call the [close, page 419](#) method when the invoke is complete. Always call [close, page 419](#) before rollback or commit. Connections or resources that are open or in use, and are not needed for commit or rollback, should be cleaned up at this point. For example, if a query was performed on a connection but no updates were performed, close the query now.
- Runtime commit or rollback—If the object was invoked, call [close, page 419](#) first, and later call either [commit, page 425](#) or [rollback, page 428](#). Call [commit, page 421](#) to commit on any connections where updates occurred, or call [rollback, page 428](#) to roll back all changes; after that, [close, page 419](#) or clean up all remaining connections and resources.

Threading

The [close, page 419](#) method can be called concurrently with any other call such as [invoke, page 439](#) or [getOutputValues, page 437](#). In such cases, any pending methods should immediately throw a [CustomProcedureException, page 423](#).

Method Detail

commit

```
public void commit()
```

This method commits an open transaction.

Throws

This method throws a [CustomProcedureException, page 423](#) if invoked for the parent transaction. It throws a `SQLException` if an error occurs.

getDescription

```
public String getDescription()
```

This method is called during data source introspection, and gets the description of the procedure. This method should not return `NULL`.

Returns

This method returns a description of the procedure.

getName

```
public String getName()
```

This method gets the short name of the procedure. This method is called during data source introspection. The short name can be overridden during data source configuration.

This method should not return NULL.

Returns

This method returns the short name of the procedure.

initialize

```
public void initialize(ExecutionEnvironment qenv)
```

This method is called once immediately after constructing the class, and initializes the query execution environment ([ExecutionEnvironment](#), [page 424](#)). The execution environment contains methods that are executed to interact with the server.

Parameter

qenv—Query execution environment.

rollback

```
public void rollback()
```

This method rolls back an open transaction.

Throws

This method throws [CustomProcedureException](#), [page 423](#), if invoked for the parent transaction. It throws `SQLException` if an error occurs.

CustomProcedureException

This exception is thrown by the methods of the extended APIs in the package `com.compositesw.extension`. For a summary of the extended APIs, see [Interface Summary, page 417](#).

```
public class CustomProcedureException
```

This exception extends `Exception`.

Constructor Summary

[CustomProcedureException, page 423](#)

[CustomProcedureException, page 423](#)

[CustomProcedureException, page 423](#)

[CustomProcedureException, page 424](#)

Constructor Detail

CustomProcedureException

```
public CustomProcedureException()
```

This is an empty constructor.

CustomProcedureException

```
public CustomProcedureException(String message)
```

This exception is thrown with a description of the error.

Parameter

`message`—Description of the error.

CustomProcedureException

```
CustomProcedureException(String message, Throwable cause)
```

This exception is thrown with descriptions of the error and the error's cause.

Parameters

message—Description of the error.

cause—Explanation of what caused the error.

CustomProcedureException

CustomProcedureException(Throwable cause)

This exception is thrown with a description of the error’s cause.

Parameter

cause—Explanation of what caused the error.

ExecutionEnvironment

ExecutionEnvironment provides an interface between a custom procedure and the TDV Server.

public interface ExecutionEnvironment

| Method Summary | |
|----------------------|---|
| void | commit, page 425 |
| ExecutionEnvironment | createTransaction, page 425 |
| java.sql.ResultSet | executeQuery, page 425 |
| int | executeUpdate, page 426 |
| String | getProperty, page 426 |
| void | log, page 427 |
| ProcedureReference | lookupNextHook, page 427 |
| ProcedureReference | lookupProcedure, page 427 |
| void | rollback, page 428 |

Method Detail

commit

`public void commit()`

This method commits an open transaction.

Throws

This method throws [CustomProcedureException, page 423](#) if invoked for the parent transaction; it throws `SQLException` if an error occurs during the commit.

createTransaction

`public ExecutionEnvironment createTransaction(int flags)`

This method starts an independent transaction, letting custom procedures have multiple independent transactions open at the same time.

Parameter

flags—Used to pass in transaction options for compensate mode, recovery mode, and recovery level.

Legal flag values are:

`COMPENSATE* | NO_COMPENSATE`

`ROLLBACK* | BEST_EFFORT`

`IGNORE_INTERRUPT* | LOG_INTERRUPT | FAIL_INTERRUPT`

Asterisks indicate the default values used if no flags are specified.

executeQuery

`public java.sql.ResultSet executeQuery (String sql, Object[] args)`

This method is used to execute a `SELECT` statement from inside the stored procedure. It should not return `NULL`.

Parameters

sql—SQL statement.

args—Arguments for the query. Can be `NULL` if there are no arguments.

The args objects should comply with the Java to SQL typing conventions listed in [Types, page 438](#). Input cursors are accepted as [CustomCursor, page 418](#) and `java.sql.ResultSet`.

Throws

This method throws [CustomProcedureException, page 423](#) or `SQLException`.

executeUpdate

```
public int executeUpdate (String sql)
```

This method executes an INSERT, UPDATE, or DELETE statement from inside the stored procedure call.

Parameter

sql—SQL statement to execute.

Returns

Number of rows affected; -1 if number of rows affected is unknown.

Throws

This method throws [CustomProcedureException, page 423](#) if there is a problem executing the SQL.

getProperty

```
public String getProperty(String name)
```

This method gets environmental properties.

Parameter

name—Property to get.

Four property options are available: `userName`, `userDomain`, `caseSensitive` and `ignoreTrailingSpaces`. Property names are not case-sensitive.

Returns

This method returns `NULL` if the property is not defined.

log

```
public void log(int level, String log_message)
```

This method sends an entry to the system log.

Parameters

level—ERROR, INFO, or DEBUG

log_message—Log entry.

lookupNextHook

```
public ProcedureReference lookupNextHook()
```

This method is used by hook procedures to invoke the next hook in the list. It should not return NULL.

Throws

This method throws [CustomProcedureException](#), page 423.

lookupProcedure

```
public lookupProcedure (String procedureName)
```

This method looks up a procedure reference from the query.

Call the [close](#), page 419 method on the returned procedure when it is no longer needed. This method does not return NULL.

Parameter

procedureName—Name of the procedure to look up.

Throws

This method throws [CustomProcedureException](#), page 423 if the procedure is not found.

rollback

public void rollback()

This method rolls back an open transaction.

Throws

This method throws [CustomProcedureException, page 423](#) if invoked for the parent transaction, or `SQLException` if an error occurs.

ParameterInfo

This class retrieves the description of procedures' input and output parameters.
public class ParameterInfo

Constructor Summary

[ParameterInfo, page 429](#) (String name, int type)
Creates a new ParameterInfo with the specified parameter values.

[ParameterInfo, page 429](#) (String name, int type, int direction)

[ParameterInfo, page 429](#) (String name, int type, int direction, ParameterInfo[] columns)

[ParameterInfo, page 430](#) (String name, int type, int direction, String xmlSchema, String localName, String namespaceURI)

Method Summary

| | |
|-----------------|--|
| ParameterInfo[] | getColumns, page 430 |
| int | getDirection, page 430 |
| String | getName, page 431 |
| int | getType, page 431 |
| String | getXmlSchema, page 431 |

Constructor Detail

ParameterInfo

public ParameterInfo (String name, int type)

Creates a new ParameterInfo with the specified parameter values.

Parameters

name—Name of the column or parameter.

type—One of the java.sql.Types: XML_STRING, TYPED_CURSOR, and GENERIC_CURSOR.

ParameterInfo

public ParameterInfo (String name, int type, int direction)

Creates a new ParameterInfo with the specified parameter values.

Parameters

name—Name of the column or parameter.

type—Types are from java.sql.Types, plus XML_STRING, TYPED_CURSOR, and GENERIC_CURSOR.

direction—The direction can be [DIRECTION_IN](#), [page 433](#), [DIRECTION_INOUT](#), [page 433](#), or [DIRECTION_OUT](#), [page 433](#). This value is passed as zero for column definitions.

ParameterInfo

public ParameterInfo (String name, int type, int direction, ParameterInfo[] columns)

Creates a new ParameterInfo with the specified parameter values.

Parameters

name—Name of the column or parameter.

type—Types are from java.sql.Types, plus XML_STRING, TYPED_CURSOR, and GENERIC_CURSOR.

direction—The direction can be [DIRECTION_IN, page 433](#), [DIRECTION_INOUT, page 433](#), or [DIRECTION_OUT, page 433](#). This value is passed as zero for a column definition.

columns—Non-null if the type is TYPED_CURSOR.

ParameterInfo

```
public ParameterInfo (String name, int type, int direction,
String xmlSchema, String localName,
String namespaceURI)
```

Creates a new ParameterInfo with the specified parameter values.

Parameters

name—Name of the column or parameter.

type—Types are from java.sql.Types, plus XML_STRING, TYPED_CURSOR, and GENERIC_CURSOR.

direction—The direction can be [DIRECTION_IN, page 433](#), [DIRECTION_INOUT, page 433](#), or [DIRECTION_OUT, page 433](#). This value is passed as zero for column definitions.

xmlSchema—Non-null if the type is XML_STRING.

localName—Local name (element name) of the selected element.

namespaceURI—URI of the namespace for the selected element.

Method Detail

getColumns

```
public ParameterInfo[] getColumns()
```

This method retrieves columns.

Returns

This method returns columns if the column data type is TYPED_CURSOR.

getDirection

```
public int getDirection()
```

This method gets the direction of the parameter.

Returns

This method returns the direction of the parameter, which can be [DIRECTION_IN](#), [page 433](#), [DIRECTION_INOUT](#), [page 433](#), or [DIRECTION_OUT](#), [page 433](#).

getName

```
public String getName()
```

This method gets the name of the column or parameter.

Returns

This method returns the name of the column or parameter.

getType

```
public int getType()
```

This method gets the type of the column or parameter.

Returns

This method returns the type of the column or parameter. The types are from `java.sql.Types`, plus `XML_STRING`, `TYPED_CURSOR`, and `GENERIC_CURSOR`.

getXmlSchema

```
public String getXmlSchema()
```

This method gets a schema.

Returns

This method returns the schema if the type is `XML_STRING`.

ProcedureConstants

This interface implements the constants that are used in the interfaces of the `com.compositesw.extension` package.

public interface ProcedureConstants

For a summary of the extended APIs, see [Interface Summary, page 417](#).

| Field Summary | |
|---------------|--|
| int | DIRECTION_IN, page 433 |
| int | DIRECTION_INOUT, page 433 |
| int | DIRECTION_NONE, page 433 |
| int | DIRECTION_OUT, page 433 |
| int | GENERIC_CURSOR, page 433 |
| int | HOOK_TYPE_SQL, page 433 |
| int | HOOK_TYPE_PROCEDURE, page 433 |
| int | LOG_ERROR, page 434 |
| int | LOG_INFO, page 434 |
| int | TXN_BEST_EFFORT, page 434 |
| int | TXN_COMPENSATE, page 434 |
| int | TXN_NO_COMPENSATE, page 435 |
| int | TXN_ROLLBACK, page 435 |
| int | TXN_IGNORE_INTERRUPT, page 434 |
| int | TXN_LOG_INTERRUPT, page 434 |
| int | TXN_NO_COMPENSATE, page 435 |
| int | TYPED_CURSOR, page 435 |
| int | XML_STRING, page 435 |

Field Detail

DIRECTION_IN

public static final int **DIRECTION_IN**

IN parameter direction constant.

DIRECTION_INOUT

public static final int **DIRECTION_INOUT**

INOUT parameter direction constant.

DIRECTION_NONE

public static final int **DIRECTION_NONE** = 0

NONE parameter direction constant.

This constant is used for [ParameterInfo, page 429](#) objects that represent columns in a cursor. See [ProcedureReference.getParameterInfo, page 439](#).

DIRECTION_OUT

public static final int **DIRECTION_OUT**

OUT parameter direction constant.

GENERIC_CURSOR

public static final int **GENERIC_CURSOR** = 5520;

Type constant for a cursor whose schema is resolved at runtime.

HOOK_TYPE_SQL

public static final int **HOOK_TYPE_SQL** = **HOOK_TYPE_SQL**

Indicates that a hook is being executed for a query or update.

HOOK_TYPE_PROCEDURE

public static final int **HOOK_TYPE_PROCEDURE** = **HOOK_TYPE_PROCEDURE**

Indicates that a hook is being executed for a stored procedure.

LOG_DEBUG

public static final int LOG_DEBUG

Debug logging level (3).

LOG_ERROR

public static final int LOG_ERROR

Error logging level (1).

LOG_INFO

public static final int LOG_INFO

Info logging level (2).

TXN_BEST_EFFORT

public static final int TXN_BEST_EFFORT

Best-effort transaction flag.

TXN_COMPENSATE

public static final int TXN_COMPENSATE = TXN_COMPENSATE

Compensate transaction flag.

TXN_FAIL_INTERRUPT

public static final int TXN_FAIL_INTERRUPT

Fail-interrupt transaction flag.

TXN_IGNORE_INTERRUPT

public static final int TXN_IGNORE_INTERRUPT

Ignore-interrupt transaction flag.

TXN_LOG_INTERRUPT

public static final int TXN_LOG_INTERRUPT

Log-interrupt transaction flag.

TXN_NO_COMPENSATE
public static final int TXN_NO_COMPENSATE

No-compensation transaction flag.

TXN_ROLLBACK
public static final int TXN_ROLLBACK

Rollback transaction flag.

TYPED_CURSOR
public static final int TYPED_CURSOR = 5521;

Type constant for a cursor with accompanying metadata.

XML_STRING
public static final int XML_STRING = 5500;

Type constant for hierarchical XML data.

ProcedureReference

The ProcedureReference interface provides a way to invoke a procedure and fetch its output values. It also provides metadata information for the procedure parameters.
public interface ProcedureReference

ProcedureReference is a parent interface for the [CustomProcedure, page 420](#) interface. It is also used as the return type when looking up a procedure from the query engine.

The type of each Java object must be the default Java object type corresponding to the input or output parameter’s SQL type, following the mapping for built-in types specified in the JDBC specification (per the getObject method on java.sql.ResultSet).

| Method Summary | |
|----------------|----------------------------------|
| void | cancel, page 436 |
| void | close, page 436 |

| Method Summary | |
|-----------------|---|
| int | getNumAffectedRows , page 436 |
| Object | getOutputValue , page 437 |
| Object[] | getOutputValues , page 437 |
| ParameterInfo[] | getParameterInfo , page 439 |
| void | invoke , page 439 |

Method Detail

cancel

void cancel()

This method cancels the procedure reference and any underlying cursors and statements.

close

public void close()

The implementation of this method should close all open cursors and all independent transactions that this method has created.

This method is called when a procedure reference is no longer needed. It is possible to call this method concurrently with any other call such as [invoke](#), [page 439](#) or [getOutputValues](#), [page 437](#), but when called concurrently with another call, this method should cause a [CustomProcedureException](#), [page 423](#).

getNumAffectedRows

public int getNumAffectedRows()

This method retrieves the number of rows that were inserted, updated, or deleted during the execution of a procedure.

Returns

A return value of -1 indicates that the number of affected rows is unknown.

Throws

This method throws [CustomProcedureException, page 423](#), or `SQLException` if an error occurs when getting the number of affected rows.

getOutputValue

```
public Object[] getOutputValue (int index)
```

This method retrieves the output value at the given index.

Returns

This method returns a procedure's output value at a given index. An output cursor can be returned as either [CustomCursor, page 418](#), or `java.sql.ResultSet`. The returned objects should comply with the Java-to-SQL typing conventions listed in [Types, page 438](#).

This method should not return `NULL`.

Throws

This method throws [CustomProcedureException, page 423](#), or `SQLException` if an error occurs when getting the output value. This method throws `ArrayIndexOutOfBoundsException` if the index value is out of bounds.

getOutputValues

```
public Object[] getOutputValues()
```

This method retrieves output values.

Returns

This method returns a procedure's output values as either [CustomCursor, page 418](#) or `java.sql.ResultSet`. The returned objects should comply with the Java-to-SQL typing conventions listed in [Types, page 438](#).

This method should not return `NULL`.

Throws

This method throws [CustomProcedureException, page 423](#), or `SQLException` if an error occurs when getting the output values.

Types

The [getOutputValues](#), page 437, method of the [ProcedureReference](#), page 435, interface retrieves the output values in a procedure. The returned objects should comply with the Java-to-SQL typing conventions as defined in this section.

The type of each Java object must be the default Java object type corresponding to the input or output parameter’s TDV JDBC data type, following the mapping for built-in types specified in the JDBC specification (per the getObject method on java.sql.ResultSet).

The following table maps the Java object types to TDV JDBC data types.

| Java Object Type | TDV JDBC Data Type |
|----------------------|-------------------------------------|
| byte[] | BINARY, VARBINARY, or LONGVARBINARY |
| java.lang.Boolean | BIT or BOOLEAN |
| java.lang.Double | DOUBLE |
| java.lang.Float | REAL or FLOAT |
| java.lang.Integer | INTEGER, SMALLINT, or TINYINT |
| java.lang.Long | BIGINT |
| java.lang.String | CHAR, VARCHAR, or LONGVARCHAR |
| java.math.BigDecimal | NUMERIC or DECIMAL |
| java.sql.Blob | BLOB |
| java.sql.Clob | CLOB |
| java.sql.Date | DATE |
| java.sql.Time | TIME |
| java.sql.TimeStamp | TIMESTAMP |

Special Types and Value

If the input or output parameter type is XML_STRING, the Java object type should be java.lang.String.

If the parameter type is `TYPED_CURSOR` or `GENERIC_CURSOR`, the Java object type is always `java.sql.ResultSet` for input parameters, and can be either [CustomCursor](#), [page 418](#), or `java.sql.ResultSet` for output parameters.

If the value is a SQL NULL, the procedure returns a Java NULL.

Hierarchical Data

This interface is primarily designed around tabular data. A stored procedure that has hierarchical input or output should accept or return one or more scalar parameters that contain XML string data. For methods that use `java.sql.Types`, the constant `XML_STRING`, [page 435](#), should be used for hierarchical XML data.

Cursors

The types `TYPED_CURSOR`, [page 435](#), and `GENERIC_CURSOR`, [page 433](#), are used to pass in and out cursor values. A typed cursor has a schema. A generic cursor's schema is resolved at run time. Procedures with generic cursor outputs cannot be used in SQL.

getParameterInfo

```
public ParameterInfo[] getParameterInfo()
```

This method is called during introspection to get the description of the procedure's input and output parameters. This method should not return NULL.

Returns

This method returns the description of the procedure's input and output parameters.

invoke

```
public void invoke(Object[] inputValues)
```

This method is called to invoke a procedure. It is called only once per procedure instance.

Parameter

`inputValues`—Values for the input parameters. Must not be NULL.

Throws

This method throws [CustomProcedureException, page 423](#), or `SQLException` if an error occurs during invocation.

Data Type Mappings for Data Sources

This topic contains tables that show the mapping of data types when different data sources are accessed through JDBC. The final main topic contains tables that list native data types supported for storing cache data.

- [TDV Data Source to JDBC Data Types, page 442](#)
- [DataDirect Mainframe to TDV Data Types, page 443](#)
- [DB2 to TDV Data Types, page 444](#)
- [File - Cache to TDV Data Types, page 446](#)
- [File - Delimited to TDV Data Types, page 446](#)
- [Greenplum to TDV Data Types, page 446](#)
- [HBase to TDV Data Types, page 449](#)
- [HSQLDB Database to TDV Data Types, page 450](#)
- [Impala to TDV Data Types, page 451](#)
- [Informix to TDV Data Types, page 453](#)
- [LDAP to TDV Data Types, page 454](#)
- [Microsoft Access to TDV Data Types, page 455](#)
- [Microsoft Excel to TDV Data Types, page 455](#)
- [Microsoft SQL Server to TDV Data Types, page 456](#)
- [MySQL to TDV Data Types, page 458](#)
- [Neoview to TDV Data Types, page 460](#)
- [Netezza to TDV Data Types, page 461](#)
- [OData to TDV Data Types, page 463](#)
- [Oracle to TDV Data Types, page 464](#)
- [ParStream to TDV Data Types, page 470](#)
- [PostgreSQL to TDV Data Types, page 472](#)
- [Redshift Data Types, page 475](#)
- [SAP HANA Data Types, page 477](#)
- [Sybase IQ to TDV Data Types, page 479](#)
- [Teradata to TDV Data Types, page 481](#)

- [Vertica to TDV Data Types, page 483](#)
- [Cache Data Type Mapping, page 485](#)

TDV Data Source to JDBC Data Types

The following table shows the data type mappings when a TDV data source is accessed through JDBC.

| Data Type | JDBC Data Type |
|-------------|----------------|
| BIGINT | BIGINT |
| BINARY | BINARY |
| BIT | BIT |
| BLOB | BLOB |
| BOOLEAN | BOOLEAN |
| CHAR | CHAR |
| CLOB | CLOB |
| DATE | DATE |
| DATETIME | TIMESTAMP |
| DECIMAL | DECIMAL |
| DOUBLE | DOUBLE |
| FLOAT | FLOAT |
| INTEGER | INTEGER |
| LONGVARCHAR | LONGVARCHAR |
| NUMERIC | NUMERIC |
| REAL | REAL |
| SMALLINT | SMALLINT |
| TIME | TIME |

| Data Type | JDBC Data Type |
|-----------|----------------|
| TINYINT | TINYINT |
| VARBINARY | VARBINARY |
| VARCHAR | VARCHAR |

DataDirect Mainframe to TDV Data Types

The following table shows the mapping from DataDirect Mainframe data types to TDV data types.

| Data Direct Mainframe Data Type | TDV Data Type |
|---------------------------------|---------------|
| BINARY | BINARY |
| CHAR | CHAR |
| CLOB | CLOB |
| DATE | DATE |
| DECIMAL | DECIMAL |
| DOUBLE | DOUBLE |
| FLOAT | FLOAT |
| INTEGER | INTEGER |
| LONGVARBINARY | BLOB |
| LONGVARCHAR | CLOB |
| MIME | BLOB |
| NUMERIC | NUMERIC |
| SMALLINT | SMALLINT |
| TIMESTAMP | TIMESTAMP |

| Data Direct Mainframe Data Type | TDV Data Type |
|---------------------------------|---------------|
| VARBINARY | VARBINARY |
| VARCHAR | VARCHAR |

DB2 to TDV Data Types

The following limits apply to DB2 data type mapping:

- The maximum length for BINARY, CHAR, VARBINARY, and VARCHAR is 32000.
- The minimum length for BINARY, CHAR, VARBINARY, and VARCHAR is 1.
- The maximum precision (p) is 38.
- The maximum scale (s) for CAST functions is 38.
- When a DECIMAL/NUMERIC data type has a precision greater than 38, it is mapped to the DB2 DOUBLE data type.

The following table shows the mapping from DB2 data types to TDV data types.

| DB2 Data Type | TDV Data Type |
|-------------------|---------------|
| BIGINT | BIGINT |
| BLOB | BLOB |
| CHAR | CHAR |
| CHAR FOR BIT DATA | BINARY |
| CHARACTER | CHAR |
| CHARACTER VARYING | VARCHAR |
| CLOB | CLOB |
| DATE | DATE |
| DBCLOB | CLOB |

| DB2 Data Type | TDV Data Type |
|---------------------------|---------------------------------|
| DECIMAL | DECIMAL |
| DOUBLE | DOUBLE |
| FLOAT | FLOAT |
| FLOAT(1) – FLOAT(21) | FLOAT [on z/OS platforms only] |
| FLOAT(22) – FLOAT(53) | DOUBLE [on z/OS platforms only] |
| GRAPHIC | CHAR |
| INTEGER | INTEGER |
| LONG VARCHAR | CLOB |
| LONG VARCHAR FOR BIT DATA | BLOB |
| LONG VARGRAPHIC | CLOB |
| LONGVAR | CLOB |
| LONGVARG | CLOB |
| REAL | REAL |
| ROWID | BINARY |
| SMALLINT | SMALLINT |
| TIME | TIME |
| TIMESTAMP | TIMESTAMP |
| VARCHAR | VARCHAR |
| VARCHAR FOR BIT DATA | VARBINARY |
| VARGRAPH | VARCHAR |
| VARGRAPHIC | VARCHAR |
| XML | XML |
| XMLCLOB | XML |

| DB2 Data Type | TDV Data Type |
|---------------|---------------|
| XMLFILE | XML |
| XMLVARCHAR | XML |

File - Cache to TDV Data Types

The following restrictions apply to four file cache data types (BINARY, VARBINARY, CHAR, and VARCHAR) when they are mapped to TDV data types:

- The maximum length is 2147483647.
- The minimum length is 1.

As of TDV 7.0, the BOOLEAN file - cache data type maps to the BOOLEAN TDV data type. For details, see [Mapping of Native to TDV Data Types Across TDV Versions, page 524](#).

File - Delimited to TDV Data Types

The following table shows the mapping from a delimited (comma-separated values or “CSV”) file data type to a TDV data type.

| CSV Flat File Data Type | TDV Data Type |
|-------------------------|---------------|
| STRING | VARCHAR |

Greenplum to TDV Data Types

This section provides the data type mappings from Greenplum to TDV data types.

Unsupported Data Types

Functions are not supported for operations on the following data types, which are mapped but not verified by TDV: CID, CIDR, INET, LINE, LSEG, MACADDR, PATH, POINT, POLYGON. For example, POINT should have a format like number,number. If a value with another format is inserted, an exception is thrown.

Type Promotion in Greenplum

In some circumstances, Greenplum performs type promotion that causes results to differ between push and no-push query execution. For example, with arithmetic operators a FLOAT4 column is converted to a FLOAT8/DOUBLE data type, and the Greenplum results have extra digits in the mantissa.

Data Type Mapping

The following table shows the data type mapping from Greenplum data types to TDV data types.

| Greenplum Data Type | TDV Data Type |
|---------------------|---|
| BIGINT | BIGINT |
| BIGSERIAL | BIGINT |
| BIT | CHAR |
| BOOL | BOOLEAN (See Mapping of Native to TDV Data Types Across TDV Versions, page 524.) |
| BOOLEAN | BOOLEAN (See Mapping of Native to TDV Data Types Across TDV Versions, page 524.) |
| BOX | VARCHAR |
| BYTEA | BLOB |
| CHAR | CHAR |
| CHARACTER | CHAR |
| CHARACTER VARYING | VARCHAR |
| CID | CHAR |
| CIDR | VARCHAR |

| Greenplum Data Type | TDV Data Type |
|---------------------|---------------|
| CIRCLE | VARCHAR |
| DATE | DATE |
| DECIMAL | DECIMAL |
| DOUBLE PRECISION | DOUBLE |
| FLOAT4 | FLOAT |
| FLOAT8 | DOUBLE |
| INET | VARCHAR |
| INT2 | SMALLINT |
| INT4 | INTEGER |
| INT8 | BIGINT |
| INTEGER | INTEGER |
| INTERVAL | VARCHAR |
| LINE | VARCHAR |
| LSEG | VARCHAR |
| MACADDR | VARCHAR |
| MONEY | DECIMAL |
| NUMERIC | DECIMAL |
| OID | BLOB |
| PATH | VARCHAR |
| POINT | CHAR |
| POLYGON | VARCHAR |
| REAL | REAL |
| SERIAL | INTEGER |

| Greenplum Data Type | TDV Data Type |
|---------------------|---------------|
| SMALLINT | SMALLINT |
| TEXT | CLOB |
| TIME | TIME |
| TIMESTAMP | TIMESTAMP |
| UUID | CHAR |
| VARBIT | VARCHAR |
| XID | INTEGER |
| XML | XML |

HBase to TDV Data Types

HBase is Apache's Hadoop database. This section provides the data type mappings from HBase to TDV data types.

Note: The maximum CAST function is 2,147,483,647

| HBase Data Type | TDV Data Type |
|------------------------|---------------|
| BIGINT | BIGINT |
| BINARY | BINARY |
| BLOB | BLOB |
| BOOLEAN | BOOLEAN |
| CHAR | CHAR |
| CHARACTER | CHAR |
| CHARACTER_LARGE_OBJECT | LONGVARCHAR |
| CHARACTER_VARYING | VARCHAR |
| CLOB | CLOB |
| DATE | DATE |

| HBase Data Type | TDV Data Type |
|-----------------|---------------|
| DECIMAL | DECIMAL |
| DOUBLE | DOUBLE |
| FLOAT | DOUBLE |
| INT | INTEGER |
| LONGVARCHAR | LONGVARCHAR |
| NUMERIC | DECIMAL |
| REAL | DOUBLE |
| SMALLINT | SMALLINT |
| TIME | TIME |
| TIMESTAMP | TIMESTAMP |
| TINYINT | TINYINT |
| VARBINARY | VARBINARY |
| VARCHAR | VARCHAR |

HSQLDB Database to TDV Data Types

This section provides the data type mappings from HSQLDB to TDV data types.

| HSQLDB Data Type | TDV Data Type |
|------------------|---------------|
| TINYINT | TINYINT |
| SMALLINT | SMALLINT |
| INT | INTEGER |
| BIGINT | BIGINT |
| NUMERIC | DECIMAL |

| HSQLDB Data Type | TDV Data Type |
|------------------------|---------------|
| DECIMAL | DECIMAL |
| BOOLEAN | BOOLEAN |
| REAL | DOUBLE |
| FLOAT | DOUBLE |
| DOUBLE | DOUBLE |
| CLOB | CLOB |
| CHAR | CHAR |
| CHARACTER | CHAR |
| CHARACTER_VARYING | VARCHAR |
| VARCHAR | VARCHAR |
| CHARACTER_LARGE_OBJECT | LONGVARCHAR |
| LONGVARCHAR | LONGVARCHAR |
| BINARY | BINARY |
| VARBINARY | VARBINARY |
| BLOB | BLOB |
| DATE | DATE |
| TIME | TIME |
| TIMESTAMP | TIMESTAMP |

Impala to TDV Data Types

This section discusses the mapping of Cloudera Impala data types to TDV data types.

The following table lists the primitive types supported in TDV Impala data sources. Any operations that involve these types are pushed to the Impala system. Types are associated with the columns in the tables.

| Primitive Type | Details |
|------------------------|--|
| Boolean | BOOLEAN—TRUE or FALSE. |
| Floating-point numbers | FLOAT—Single precision. DOUBLE—Double precision. |
| Integer types | TINYINT—1-byte integer. SMALLINT—2-byte integer. INT—4-byte integer. BIGINT—8-byte integer. |
| String type | STRING—Sequence of characters in a specified character set. |

The following complex types (and the operators and functions that use these types) are available in Impala systems but are **not** supported in TDV Impala data sources:

- Structs
- Maps
- Arrays

So that Impala data sources can interact with other TDV data sources, Impala data types are mapped to the following TDV data types.

| Impala Data Type | TDV Data Type |
|------------------|---|
| BIGINT | BIGINT |
| BOOLEAN | BOOLEAN (See Mapping of Native to TDV Data Types Across TDV Versions , page 524.) |
| DOUBLE | DOUBLE |
| FLOAT | DOUBLE |
| INT | INTEGER |
| SMALLINT | SMALLINT |

| Impala Data Type | TDV Data Type |
|------------------|---|
| STRING | LONGVARCHAR Maximum length for CAST: 2147483647. |
| TINYINT | TINYINT |

Informix to TDV Data Types

Informix large-object types (“*LOBs”) are mapped to BLOBS or CLOBs.

The following table shows the mapping from Informix data types to TDV data types.

| Informix Data Type | TDV Data Type |
|--------------------|---|
| BLOB | BLOB |
| BOOLEAN | BOOLEAN (See Mapping of Native to TDV Data Types Across TDV Versions , page 524.) |
| BYTE | VARBINARY |
| CHAR | CHAR |
| CHARACTER | CHAR |
| CLOB | CLOB |
| DATE | DATE |
| DATETIME | TIMESTAMP |
| DEC | DECIMAL |
| DECIMAL | DECIMAL |
| DOUBLE PRECISION | DOUBLE |
| FLOAT | DOUBLE |
| INT | INTEGER |

| Informix Data Type | TDV Data Type |
|--------------------|---------------|
| INT8 | LONG |
| INTEGER | INTEGER |
| LVARCHAR | CLOB |
| MONEY | DECIMAL |
| NCHAR | CHAR |
| NULL | NULL |
| NUMERIC | NUMERIC |
| NVARCHAR | VARCHAR |
| REAL | REAL |
| SERIAL | INTEGER |
| SERIAL8 | LONG |
| SMALLFLOAT | FLOAT |
| SMALLINT | SMALLINT |
| TEXT | CLOB |
| VARCHAR | VARCHAR |

LDAP to TDV Data Types

The following table shows the mapping from a LDAP data type to a TDV data type.

| LDAP Data Type | TDV Data Type |
|----------------|---------------|
| OCTET STRING | VARCHAR |

Microsoft Access to TDV Data Types

The following table shows the mapping from Microsoft Access data types to TDV data types.

| Microsoft Access Data Type | TDV Data Type |
|----------------------------|---------------|
| BIT | BIT |
| BYTE | TINYINT |
| COUNTER | INTEGER |
| CURRENCY | NUMERIC |
| DATETIME | TIMESTAMP |
| DECIMAL | DECIMAL |
| DOUBLE | DOUBLE |
| FLOAT | DOUBLE |
| INTEGER | INTEGER |
| LONGBINARY | BLOB |
| LONGCHAR | CLOB |
| REAL | REAL |
| SMALLINT | SMALLINT |
| VARCHAR | VARCHAR |

Microsoft Excel to TDV Data Types

The following table shows the mapping from Microsoft Excel data types to TDV data types.

Note: The NUMBER data types returned from the JDBC/ODBC driver do not accurately reflect the real precision and scale if you have formatted the cells in Excel with the following categories: NUMBER, PERCENTAGE, SCIENTIFIC, and FRACTION.

| Microsoft Excel Data Type | TDV Data Type |
|---------------------------|----------------|
| BIT | BIT |
| BIGINT | BIGINT |
| CURRENCY | DOUBLE |
| DATETIME | TIMESTAMP |
| NUMBER | DOUBLE |
| VARCHAR | VARCHAR(32676) |

Microsoft SQL Server to TDV Data Types

The following table shows the mapping from SQL Server data types to TDV data types.

| Microsoft SQL Server Data Type | TDV Data Type | Notes |
|--------------------------------|---------------|---------------------------|
| BIGINT | BIGINT | |
| BINARY | BINARY | |
| BIT | BIT | |
| CHAR | CHAR | |
| DATE | DATE | SQL Server 2008 and 2012. |
| DATETIME | TIMESTAMP | |
| DATETIME2 | TIMESTAMP | SQL Server 2008 and 2012. |
| DATETIME2(0) – DATETIME2(7) | TIMESTAMP | SQL Server 2008 and 2012. |
| DATETIMEOFFSET | VARCHAR | SQL Server 2008 and 2012. |

| Microsoft SQL Server Data Type | TDV Data Type | Notes |
|---------------------------------------|---------------|---|
| DATETIMEOFFSET(0) – DATETIMEOFFSET(7) | VARCHAR | SQL Server 2008 and 2012. |
| DECIMAL | DECIMAL | |
| FLOAT | DOUBLE | |
| IMAGE | BLOB | |
| INT | INTEGER | |
| INT IDENTITY | INTEGER | |
| MONEY | DECIMAL | |
| NCHAR | CHAR | |
| NTEXT | CLOB | |
| NUMERIC | NUMERIC | |
| NVARCHAR | VARCHAR | |
| REAL | FLOAT | |
| SMALLDATETIME | TIMESTAMP | |
| SMALLINT | SMALLINT | |
| SMALLMONEY | DECIMAL | |
| SQL_VARIANT | OTHER | ODBC does not fully support this data type. |
| TABLE | OTHER | |
| TEXT | CLOB | |
| TIME | TIME | SQL Server 2008 and 2012. |
| TIME() – TIME(7) | TIME | SQL Server 2008 and 2012. |
| TIMESTAMP | VARBINARY | |
| TINYINT | SMALLINT | |

| Microsoft SQL Server Data Type | TDV Data Type | Notes |
|--------------------------------|---------------|-------|
| UNIQUEIDENTIFIER | CHAR | |
| VARBINARY | VARBINARY | |
| VARCHAR | VARCHAR | |
| XML | XML | |

MySQL to TDV Data Types

The following table shows the mapping from MySQL data types to TDV data types.

Numeric scale (s) has a range of 0 through 30, but it cannot exceed precision (p). Precision has a range of: 1 through 264 (MySQL 5.0.2 and earlier); 1 through 64 (MySQL 5.0.3 to 5.0.5); or 1 through 65 (MySQL 5.0.6 and later).

| MySQL Data Type | TDV Data Type | Notes |
|-----------------|--------------------------|---|
| BIGINT | BIGINT NUMERIC(20, 0) | |
| BINARY | BINARY | |
| BIT | BIT | |
| BIT(1) | BOOLEAN | MySQL 5.0 override. |
| BLOB | VARBINARY | |
| BOOL | BOOLEAN | (See Mapping of Native to TDV Data Types Across TDV Versions , page 524.) |
| CHAR | CHAR | |
| DATE | DATE | |
| DATETIME | TIMESTAMP | |
| DEC | DECIMAL | |

| MySQL Data Type | TDV Data Type | Notes |
|------------------|---------------|-----------------------------------|
| DECIMAL | DECIMAL | |
| DOUBLE | DOUBLE | |
| DOUBLE PRECISION | DOUBLE | |
| DOUBLE UNSIGNED | DOUBLE | |
| ENUM | VARCHAR | |
| FIXED | DECIMAL | |
| FLOAT | DOUBLE | |
| FLOAT UNSIGNED | DOUBLE | |
| INT | INTEGER | Unsigned INT or unsigned INTEGER. |
| INTEGER | INTEGER | |
| LOBLOB | BLOB | |
| LONGTEXT | CLOB | |
| MEDIUMBLOB | BLOB | |
| MEDIUMINT | INTEGER | |
| MEDIUMTEXT | CLOB | |
| NUMERIC | DECIMAL | |
| REAL | DOUBLE | |
| SET | VARCHAR | |
| SMALLINT | SMALLINT | |
| TEXT | VARCHAR | |
| TIME | TIME | |
| TIMESTAMP | TIMESTAMP | |
| TINYBLOB | VARBINARY | |

| MySQL Data Type | TDV Data Type | Notes |
|-----------------|---------------|-------|
| TINYINT | TINYINT | |
| TINYTEXT | VARCHAR | |
| VARBINARY | VARBINARY | |
| VARCHAR | VARCHAR | |
| YEAR | SMALLINT | |

Neoview to TDV Data Types

Neoview does not support BINARY or BLOB data types.

The following table shows the mapping from Neoview data types to TDV data types:

| Neoview Data Type | TDV Data Type |
|-------------------|---------------|
| CHAR | CHAR |
| CHARACTER | CHAR |
| CHARACTER VARYING | VARCHAR |
| CHAR VARYING | VARCHAR |
| DATE | DATE |
| DECIMAL | DECIMAL |
| DOUBLE PRECISION | DOUBLE |
| FLOAT | FLOAT |
| INTEGER | INTEGER |
| INTEGER UNSIGNED | BIGINT |
| LARGEINT | BIGINT |
| LONG VARCHAR | VARCHAR |
| NCHAR | CHAR |

| Neoview Data Type | TDV Data Type |
|----------------------------|---------------|
| NATIONAL CHAR | CHAR |
| NATIONAL CHARACTER | CHAR |
| NCHAR VARYING | VARCHAR |
| NATIONAL CHARACTER VARYING | VARCHAR |
| NATIONAL CHAR VARYING | VARCHAR |
| NUMERIC | NUMERIC |
| REAL | REAL |
| SMALLINT | SMALLINT |
| SMALLINT UNSIGNED | INTEGER |
| TIME | TIME |
| TIMESTAMP | TIMESTAMP |
| VARCHAR | VARCHAR |

Netezza to TDV Data Types

Netezza's BIT data type is equivalent to a BOOLEAN type. However, it is not accepted in mathematical operations.

The following table shows the mapping from Netezza data types to TDV data types.

| Netezza Data Type | TDV Data Type |
|-------------------|---|
| BIGINT | BIGINT |
| BOOL or BOOLEAN | BOOLEAN (See Mapping of Native to TDV Data Types Across TDV Versions, page 524.) |
| BYTEINT | TINYINT |

| Netezza Data Type | TDV Data Type |
|---------------------|--|
| CHAR | CHAR |
| DATE | DATE |
| DECIMAL | DECIMAL |
| DOUBLE PRECISION | DOUBLE |
| FLOAT | FLOAT |
| INT | INTEGER |
| INT1 | TINYINT |
| INT2 | SMALLINT |
| INT4 | INTEGER |
| INT8 | BIGINT |
| INTEGER | INTEGER |
| INTERVAL | VARCHAR |
| NCHAR | CHAR |
| NVARCHAR | VARCHAR |
| NUMERIC | DECIMAL |
| REAL | REAL |
| SMALLINT | SMALLINT |
| TIME | TIME |
| TIMETZ | TIMESTAMP VARCHAR [not in 5.0, 6.0] |
| TIME WITH TIME ZONE | VARCHAR |
| TIMESTAMP | TIMESTAMP |
| VARCHAR | VARCHAR |

OData to TDV Data Types

This section shows the mapping from OData data types to TDV data types.

| OData Data Type | TDV Data Type |
|--------------------------|---------------|
| BINARY | BINARY |
| BOOLEAN | BOOLEAN |
| BYTE | CHAR |
| DATE | DATE |
| DATETIME | TIMESTAMP |
| DATETIMEOFFSET | TIMESTAMP |
| DECIMAL | DECIMAL |
| DOUBLE | DOUBLE |
| DURATION | VARCHAR |
| GEOGRAPHY | VARCHAR |
| GEOGRAPHYCOLLECTION | VARCHAR |
| GEOGRAPHYLINestring | VARCHAR |
| GEOGRAPHYMULTILINestring | VARCHAR |
| GEOGRAPHYMULTIPOINT | VARCHAR |
| GEOGRAPHYMULTIPOLYGON | VARCHAR |
| GEOGRAPHYPOINT | VARCHAR |
| GEOGRAPHYPOLYGON | VARCHAR |
| GEOMETRY | VARCHAR |
| GEOMETRYCOLLECTION | VARCHAR |
| GEOMETRYLINestring | VARCHAR |
| GEOMETRYMULTILINestring | VARCHAR |

| OData Data Type | TDV Data Type |
|----------------------|---------------|
| GEOMETRYMULTIPOINT | VARCHAR |
| GEOMETRYMULTIPOLYGON | VARCHAR |
| GEOMETRYPOINT | VARCHAR |
| GEOMETRYPOLYGON | VARCHAR |
| GUID | VARCHAR(36) |
| INT16 | SMALLINT |
| INT32 | INTEGER |
| INT64 | BIGINT |
| SBYTE | TINYINT |
| SINGLE | DECIMAL(32,7) |
| STREAM | BLOB |
| STRING | LONGVARCHAR |
| TIME | VARCHAR(10) |
| TIMEOFDAY | TIME |

Oracle to TDV Data Types

This section shows the mapping from Oracle data types to TDV data types.

- [Oracle NUMBER Data Types and TDV Data Types, page 465](#)
- [Oracle to Data Types Common to All Versions, page 466](#)
- [Oracle 9i to TDV Data Types, page 467](#)
- [Oracle 10g to TDV Data Types, page 468](#)
- [Oracle 11g to TDV Data Types, page 469](#)

Oracle NUMBER Data Types and TDV Data Types

Static Mapping

The following details apply to the mapping of Oracle NUMBER data types to TDV data types:

- If the scale of the NUMBER column is not specified, it is mapped as DOUBLE.
- If either the precision or the scale is NULL, the data type is mapped to DOUBLE.
- If the precision and scale are defined as nonzero values, the data type is mapped to DECIMAL.
- If the scale is 0 (zero), different precision values result in different data type mappings:
 - If the precision is less than or equal to 2, NUMBER is mapped to TINYINT.
 - If the precision is less than or equal to 4, NUMBER is mapped to SMALLINT.
 - If the precision is less than or equal to 9, NUMBER is mapped to INTEGER.
 - If the precision is less than or equal to 19, NUMBER is mapped to BIGINT.
 - Otherwise, NUMBER is mapped to NUMERIC with 0 (zero) scale.
- If the precision is not specified, it defaults to 38.
- When casting a value as DECIMAL(p, s), (for example, CAST (Oracle_column AS DECIMAL(40))):
 - Where the precision (p) is greater than 38, it is processed in TDV.
 - The maximum precision supported in TDV is Integer.MAX_VALUE, which is 2147483647.
 - The maximum scale that TDV supports is 255.
 - Any scale larger than 255 is automatically reduced to 255.

Results Mapping

Oracle has no SQL-standard equivalent of INTEGER. An INTEGER in Oracle is NUMBER(38), but division promotes it to NUMBER(p, s) where p and s are the precision and scale needed to represent the result with fractional digits. If Oracle users expect a specific precision and scale in arithmetic operations involving NUMBER or NUMBER(n) types, they should CAST the result to a suitable type, with appropriate precision and scale.

Oracle to Data Types Common to All Versions

This table lists the base Oracle data type to TDV data type mappings. Specific Oracle database versions might have additional data types. The additional data type mappings are listed in the sections for specific Oracle database versions.

Note: While Oracle honors trailing spaces in general, it ignores them when comparing CHARs. When TDV is set to honor trailing spaces, a filter on a CHAR column might return different results when executed in Oracle vs. TDV.

| Oracle Base Data Type | TDV Data Type | Notes |
|-----------------------|---------------|---|
| BFILE | BLOB | |
| BLOB | BLOB | |
| CHAR | CHAR | |
| CLOB | CLOB | |
| DATE | TIMESTAMP | |
| FLOAT | FLOAT | |
| LONG | CLOB | |
| LONG RAW | BLOB | |
| LONG VARCHAR | CLOB | |
| NCHAR | CHAR | |
| NCLOB | CLOB | |
| NUMBER | DECIMAL | See Oracle NUMBER Data Types and TDV Data Types, page 465 . |

| Oracle Base Data Type | TDV Data Type | Notes |
|-----------------------|---------------|--|
| NUMBER(2,0) | TINYINT | In these examples, a hyphen indicates that the value is not specified in Oracle. |
| NUMBER(4,0) | SMALLINT | |
| NUMBER(8,0) | INTEGER | |
| NUMBER(15,0) | BIGINT | |
| NUMBER(22,0) | NUMERIC(22,0) | |
| NUMBER(10,3) | DECIMAL(10,3) | |
| NUMBER(-,0) | NUMERIC(38,0) | |
| NUMBER(-,2) | DECIMAL(38,2) | |
| NUMBER(12,-) | DOUBLE | |
| NUMBER(-,-) | DOUBLE | |
| NVARCHAR | VARCHAR | |
| NVARCHAR2 | VARCHAR | |
| RAW | VARBINARY | |
| ROWID | VARCHAR | |
| UROWID | VARCHAR | |
| VARCHAR | VARCHAR | |
| VARCHAR2 | VARCHAR | |

Oracle 9i to TDV Data Types

The following table shows the mapping from Oracle 9i data types to TDV data types.

| Oracle 9i Data Type | TDV Data Type |
|---------------------|---------------|
| ANYDATA | OTHER |
| ANYDATASET | OTHER |

| Oracle 9i Data Type | TDV Data Type |
|--|---------------|
| ANYTYPE | OTHER |
| INTERVAL DAY(0) TO SECOND(0) – INTERVAL DAY(9) TO SECOND(9) | VARCHAR |
| INTERVAL YEAR(0) TO MONTH – INTERVAL YEAR(9) TO MONTH | VARCHAR |
| TIMESTAMP | TIMESTAMP |
| TIMESTAMP(0) | TIMESTAMP |
| TIMESTAMP(0) WITH LOCAL TIME ZONE – TIMESTAMP(9) WITH LOCAL TIME ZONE | OTHER |
| TIMESTAMP(0) WITH TIME ZONE – TIMESTAMP(9) WITH TIME ZONE | TIMESTAMP |
| TIMESTAMP(9) | TIMESTAMP |
| URITYPE | OTHER |
| UROWID | VARCHAR |
| XMLTYPE | XML |

Oracle 10g to TDV Data Types

The maximum length of VARBINARY is 2000.

The following table shows the mapping from Oracle 10g data types to TDV data types.

| Oracle 10g Data Type | TDV Data Type |
|----------------------|---------------|
| ANYDATA | OTHER |
| ANYDATASET | OTHER |
| ANYTYPE | OTHER |
| BINARY DOUBLE | DOUBLE |
| BINARY FLOAT | FLOAT |

| Oracle 10g Data Type | TDV Data Type |
|---|--|
| INTERVAL DAY(0) TO SECOND(0) – INTERVAL DAY(9) TO SECOND(9) | VARCHAR |
| INTERVAL YEAR(0) TO MONTH – INTERVAL YEAR(9) TO MONTH | VARCHAR |
| SDO_GEORASTER | OTHER |
| SI_STILLIMAGE | VARBINARY |
| TIMESTAMP | TIMESTAMP [Uses FLOOR() instead of ROUND() on the difference.] |
| TIMESTAMP(0) – TIMESTAMP(9) | TIMESTAMP |
| TIMESTAMP(0) WITH LOCAL TIME ZONE – TIMESTAMP(9) WITH LOCAL TIME ZONE | TIMESTAMP |
| TIMESTAMP(0) WITH TIME ZONE – TIMESTAMP(9) WITH TIME ZONE | TIMESTAMP |
| URITYPE | OTHER |
| UROWID | VARCHAR |
| XMLTYPE | XML |

Oracle 11g to TDV Data Types

The maximum VARBINARY length in Oracle 11g is 2000.

The following table shows the mapping from Oracle 11g data types to TDV data types.

| Oracle 11g Data Type | TDV Data Type |
|----------------------|---------------|
| ANYDATA | OTHER |
| ANYDATASET | OTHER |
| ANYTYPE | OTHER |

| Oracle 11g Data Type | TDV Data Type |
|---|---------------|
| BINARY DOUBLE | DOUBLE |
| BINARY FLOAT | FLOAT |
| INTERVAL DAY(0) TO SECOND(0) – INTERVAL DAY(9) TO SECOND(9) | VARCHAR |
| INTERVAL YEAR(0) TO MONTH – INTERVAL YEAR(9) TO MONTH | VARCHAR |
| SDO_GEORASTER | OTHER |
| SI_STILLIMAGE | VARBINARY |
| TIMESTAMP | TIMESTAMP |
| TIMESTAMP(0) – TIMESTAMP(9) | TIMESTAMP |
| TIMESTAMP(0) WITH LOCAL TIME ZONE – TIMESTAMP(9) WITH LOCAL TIME ZONE | TIMESTAMP |
| TIMESTAMP(0) WITH TIME ZONE – TIMESTAMP(9) WITH TIME ZONE | TIMESTAMP |
| URITYPE | OTHER |
| UROWID | VARCHAR |
| XMLTYPE | XML |

ParStream to TDV Data Types

The table below shows the mapping from ParStreeam data types to TDV data types.

ParStream data conversion and comparison have these traits:

- LONGVARCHAR can only contain 2147483647 characters at most. Any string beyond such length will be ignored.

- BLOB can only contain 2147483647 characters at most. Any string beyond such length will be ignored.

| ParStream Data Type | TDV Data Type | Range |
|---------------------|---------------|--|
| INT8 | TINYINT | -128 to +126 (NULL==127) |
| INT16 | SMALLINT | -32768 to +32766 (NULL==32767) |
| INT32 | INTEGER | -2147483648 to +2147483646 (NULL==2147483647) |
| INT64 | BIGINT | -9223372036854775808 to +9223372036854775806 (NULL==9223372036854775807) |
| UINT8 | SMALLINT | 0 to 254 (NULL== 255) |
| UINT16 | INTEGER | 0 to +65534 (NULL== 65535) |
| UINT32 | BIGINT | 0 to +4294967294 (NULL== 4294967295) |
| UINT64 | DECIMAL(20,0) | 0 to +18446744073709551614 (NULL==18446744073709551615) |
| FLOAT | FLOAT | |
| DOUBLE | DOUBLE | |
| DATE | DATE | 01.01.0000 to 31.12.9999 |
| SHORTDATE | DATE | 01.01.2000 to 31.12.2178 |
| TIME | TIME | 00:00:00 or 00:00:00.000 to 23:59:59.999 |

| ParStream Data Type | TDV Data Type | Range |
|------------------------------------|---------------|--|
| TIMESTAMP | TIMESTAMP | 01.01.0000 00:00:00 to 31.12.9999 23:59:59.999 |
| VARSTRING | LONGVARCHAR | 0 to ? |
| VARSTRING COMPRESSION HASH64 | OTHER | |
| BLOB | BLOB | 1024*1024 or 220 to ? |
| BLOB COMPRESSION HASH64 | OTHER | |
| BITVECTOR8 | BINARY(8) | NULL == #00000000 |
| MULTI_VALUE | LONGVARCHAR | |

PostgreSQL to TDV Data Types

The table below shows the mapping from PostgreSQL data types to TDV data types.

PostgreSQL data conversion and comparison have these traits:

- Interval years converted to months result in a TDV data type of VARCHAR.
- Interval days converted to seconds result in a TDV data type of VARCHAR.
- Timestamps with a time zone or a local time zone result in a TDV data type of TIMESTAMP.
- While PostgreSQL honors trailing spaces in general, it ignores them when comparing CHARs. When TDV is set to honor trailing spaces, a filter on a CHAR column might return different results when executed in PostgreSQL vs. TDV.

| PostgreSQL Data Type | TDV Data Type |
|----------------------|---------------|
| BIGINT | BIGINT |

| PostgreSQL Data Type | TDV Data Type |
|----------------------|---|
| BIGSERIAL | BIGINT |
| BINARY DOUBLE | DOUBLE |
| BINARY FLOAT | REAL |
| BIT | CHAR |
| BOOL | CHAR |
| BOOLEAN | BOOLEAN (See Mapping of Native to TDV Data Types Across TDV Versions , page 524.) |
| BOX | VARCHAR |
| BPCHAR | CHAR |
| BYTEA | BLOB |
| CHAR | CHAR |
| CHARACTER | CHAR |
| CHARACTER VARYING | VARCHAR |
| CIDR | VARCHAR |
| CIRCLE | VARCHAR |
| DATE | DATE |
| DATETIME | TIMESTAMP |
| DOUBLE PRECISION | DOUBLE |
| FLOAT4 | REAL |
| FLOAT8 | DOUBLE |
| INET | VARCHAR |
| INT | INTEGER |
| INT(2) | SMALLINT |

| PostgreSQL Data Type | TDV Data Type |
|----------------------|---------------|
| INT(4) | INTEGER |
| INT(8) | BIGINT |
| INTEGER | INTEGER |
| INTERVAL | VARCHAR |
| LINE | VARCHAR |
| LONG | CLOB |
| LSEG | VARCHAR |
| MACADDR | VARCHAR |
| MONEY | DECIMAL |
| NUMBER | DECIMAL |
| NUMERIC | NUMERIC |
| OID | BLOB |
| PATH | VARCHAR |
| POINT | CHAR |
| POLYGON | VARCHAR |
| REAL | REAL |
| ROWID | VARCHAR |
| SERIAL | INTEGER |
| SMALLDATETIME | TIMESTAMP |
| SMALLINT | SMALLINT |
| TEXT | CLOB |
| TIME | TIME |
| TIMESTAMP | TIMESTAMP |

| PostgreSQL Data Type | TDV Data Type |
|----------------------|---------------|
| TIMESTAMPTZ | TIMESTAMP |
| TIMETZ | TIME |
| TINYINT | SMALLINT |
| UROWID | VARCHAR |
| UUID | CHAR |
| VARBIT | VARCHAR |
| VARCHAR | VARCHAR |
| VARCHAR2 | VARCHAR |
| XID | INTEGER |
| XML | XML |

Redshift Data Types

The table below shows the mapping from Redshift data types to TDV data types.

Redshift data conversion and comparison have these traits:

- CHAR length: minimum is 1, maximum is 10485760
- VARCHAR length: minimum is 1, maximum is 10485760
- VARBINARY length: maximum is 2000
- Precision: maximum is 38

| Redshift Data Type | TDV Data Type |
|--------------------|---------------|
| BIGINT | BIGINT |
| BOOL | BOOLEAN |
| BOOLEAN | BOOLEAN |
| BPCHAR | CHAR |

| Redshift Data Type | TDV Data Type |
|--------------------|---------------|
| CHAR | CHAR |
| CHARACTER | CHAR |
| CHARACTER_VARYING | VARCHAR |
| DATE | DATE |
| DECIMAL | DECIMAL |
| DOUBLE_PRECISION | DOUBLE |
| FLOAT | FLOAT |
| FLOAT4 | REAL |
| FLOAT8 | DOUBLE |
| INT | INTEGER |
| INT2 | SMALLINT |
| INT4 | INTEGER |
| INT8 | BIGINT |
| INTEGER | INTEGER |
| NCHAR | CHAR |
| NUMERIC | DECIMAL |
| NVARCHAR | VARCHAR |
| REAL | REAL |
| SMALLINT | SMALLINT |
| TEXT | VARCHAR |
| TIMESTAMP | TIMESTAMP |
| VARCHAR | VARCHAR |

SAP HANA Data Types

The following table shows the mapping from SAP HANA data types to TDV data types.

| SAP HANA Data Type | TDV Data Type |
|--------------------|---------------|
| ALPHANUM | VARCHAR |
| BIGINT | BIGINT |
| BINARY | BINARY |
| BINTEXT | CLOB |
| BLOB | BLOB |
| CHAR | CHAR |
| CLOB | CLOB |
| DATE | DATE |
| DECIMAL | DECIMAL |
| DOUBLE | DOUBLE |
| FLOAT | DOUBLE |
| INTEGER | INTEGER |
| NCHAR | CHAR |
| NCLOB | CLOB |
| NVARCHAR | VARCHAR |
| REAL | FLOAT |
| SECONDDATE | TIMESTAMP |
| SHORTTEXT | VARCHAR |
| SMALLDECIMAL | FLOAT |
| SMALLINT | SMALLINT |

| SAP HANA Data Type | TDV Data Type |
|--------------------|---------------|
| TIME | TIME |
| TIMESTAMP | TIMESTAMP |
| TINYINT | SMALLINT |
| VARBINARY | VARBINARY |
| VARCHAR | VARCHAR |

Sybase ASE to TDV Data Types

The following table shows the mapping from Sybase ASE data types to TDV data types.

| Sybase ASE Data Type | TDV Data Type |
|----------------------|---------------|
| BINARY | BINARY |
| BIT | BIT |
| CHAR | CHAR |
| DATETIME | TIMESTAMP |
| DECIMAL | DECIMAL |
| FLOAT | DOUBLE |
| IMAGE | BLOB |
| INT | INTEGER |
| MONEY | DECIMAL |
| NCHAR | CHAR |
| NUMERIC | NUMERIC |
| NVARCHAR | VARCHAR |
| REAL | REAL |

| Sybase ASE Data Type | TDV Data Type |
|----------------------|---------------|
| SMALLDATETIME | TIMESTAMP |
| SMALLINT | SMALLINT |
| SMALLMONEY | DECIMAL |
| SYSNAME | VARCHAR |
| TEXT | CLOB |
| TIMESTAMP | VARBINARY |
| TINYINT | SMALLINT |
| UNICHAR | CHAR |
| UNIVARCHAR | VARCHAR |
| VARBINARY | VARBINARY |
| VARCHAR | VARCHAR |

Sybase IQ to TDV Data Types

The following table shows the mapping from Sybase IQ data types to TDV data types. Refer also to “Adding a Sybase Data Source” in the *TDV User Guide*.

| Sybase IQ Data Type | TDV Data Type |
|---------------------|--|
| BIGINT | TDV partially supports BIGINT by mapping it to LONG.MAX_VALUE. If a query result value exceeds +9,223,372,036,854,775,807, an error is returned. |
| BINARY | BINARY |
| BIT | BIT |
| BLOB | BLOB |
| CHAR | CHAR (Maximum length 32766.) |
| CLOB | CLOB |

| Sybase IQ Data Type | TDV Data Type |
|---------------------|---------------|
| DATETIME | TIMESTAMP |
| DECIMAL | DECIMAL |
| FLOAT | DOUBLE |
| IMAGE | BLOB |
| INT | INTEGER |
| LONG BINARY | BLOB |
| LONG VARCHAR | CLOB |
| MONEY | DECIMAL |
| NCHAR | CHAR |
| NUMERIC | NUMERIC |
| NVARCHAR | VARCHAR |
| REAL | REAL |
| SMALLDATETIME | TIMESTAMP |
| SMALLINT | SMALLINT |
| SMALLMONEY | DECIMAL |
| SYSNAME | VARCHAR |
| TEXT | CLOB |
| TIMESTAMP | TIMESTAMP |
| TINYINT | SMALLINT |
| UNICHAR | CHAR |
| UNIQUEIDENTIFIER | BINARY |
| UNIVARCHAR | VARCHAR |
| UNSIGNED INT | INTEGER |

| Sybase IQ Data Type | TDV Data Type |
|---------------------|-----------------------------------|
| VARBINARY | VARBINARY (Maximum length 32766.) |
| VARCHAR | VARCHAR (Maximum length 32766.) |

Teradata to TDV Data Types

The following table shows the mapping from Teradata data types to TDV data types.

Teradata data types have these characteristics:

- FLOAT and REAL data types are synonymous with DOUBLE PRECISION.
- For Teradata version 15, the maximum length for BINARY and VARBINARY is 64000; for CHAR and VARCHAR it is 32000; for BLOB it is 2097088000; for CLOB it is 1048544000; and for JSON it is 8388096.
- For all supported versions of Teradata except version 15, the maximum length for BINARY, CHAR, VARBINARY, and VARCHAR is 32000; and the JSON data type is not supported.
- IN operator with subquery cannot be pushed down,
- The native data types BLOB, CLOB, JSON, and XML are not supported in DISTINCT, EXCEPT, GROUP BY, HAVING, INTERSECT, JOIN ON, ORDER BY or UNION clauses,

| Teradata Data Type | TDV Data Type |
|--------------------|---------------|
| BIGINT | BIGINT |
| BLOB | BLOB |
| BYTE | BINARY |
| BYTEINT | TINYINT |
| CHAR | CHAR |
| CLOB | CLOB |
| DATE | DATE |
| DECIMAL | DECIMAL |

| Teradata Data Type | TDV Data Type |
|-----------------------------|---------------------------|
| DOUBLE PRECISION | DOUBLE |
| FLOAT | DOUBLE |
| GRAPHIC | CHAR |
| INTEGER | INTEGER |
| INTERVAL DAY | INTERVAL DAY |
| INTERVAL DAY TO HOUR | INTERVAL DAY TO HOUR |
| INTERVAL DAY TO MINUTE | INTERVAL DAY TO MINUTE |
| INTERVAL DAY TO SECOND | INTERVAL DAY TO SECOND |
| INTERVAL HOUR | INTERVAL HOUR |
| INTERVAL HOUR TO MINUTE | INTERVAL HOUR TO MINUTE |
| INTERVAL HOUR TO SECOND | INTERVAL HOUR TO SECOND |
| INTERVAL MINUTE | INTERVAL MINUTE |
| INTERVAL MINUTE TO SECOND | INTERVAL MINUTE TO SECOND |
| INTERVAL MONTH | INTERVAL MONTH |
| INTERVAL SECOND | INTERVAL SECOND |
| INTERVAL YEAR | INTERVAL YEAR |
| INTERVAL YEAR TO MONTH | INTERVAL YEAR TO MONTH |
| LONG VARCHAR | CLOB |
| NUMERIC | NUMERIC |
| PERIOD(DATE) | VARCHAR |
| PERIOD(TIME) | VARCHAR |
| PERIOD(TIMESTAMP) | VARCHAR |
| PERIOD(TIME WITH TIME ZONE) | VARCHAR |

| Teradata Data Type | TDV Data Type |
|----------------------------------|-----------------------|
| PERIOD(TIMESTAMP WITH TIME ZONE) | VARCHAR |
| REAL | DOUBLE |
| SMALLINT | SMALLINT |
| TIME | TIME |
| TIME WITH ZONE | VARCHAR |
| TIMESTAMP | TIMESTAMP |
| TIMESTAMP WITH ZONE | VARCHAR |
| VARBYTE | VARBINARY |
| VARCHAR | VARCHAR |
| VARGRAPHIC | VARCHAR |
| XML | XML (version 15 only) |

Vertica to TDV Data Types

Mapped Vertica data types have the following restrictions:

- Maximum BINARY length is 65000.
- Maximum VARBINARY length is 65000.
- Maximum CHAR length is 65000.
- Maximum VARCHAR length is 65000.

The following table shows the mapping from Vertica data types to TDV data types.

| Vertica Data Type | TDV Data Type |
|-------------------|---------------|
| BIGINT | BIGINT |
| BINARY | BINARY |

| Vertica Data Type | TDV Data Type |
|-------------------|---|
| BINARY VARYING | VARBINARY |
| BOOL | CHAR |
| BOOLEAN | BOOLEAN (See Mapping of Native to TDV Data Types Across TDV Versions, page 524.) |
| BYTEA | VARBINARY |
| CHAR | CHAR |
| CHARACTER | CHAR |
| CHARACTER VARYING | VARCHAR |
| DATE | DATE |
| DATETIME | TIMESTAMP |
| DECIMAL | DECIMAL |
| DOUBLE PRECISION | DOUBLE |
| FLOAT | DOUBLE |
| INT | BIGINT |
| INTEGER | BIGINT |
| INTERVAL | VARCHAR |
| MONEY | DECIMAL |
| NUMBER | NUMBER |
| NUMERIC | DECIMAL |
| RAW | VARBINARY |
| REAL | DOUBLE |
| SMALLDATETIME | TIMESTAMP |
| SMALLINT | BIGINT |

| Vertica Data Type | TDV Data Type |
|-------------------|---------------|
| TIME | TIME |
| TIMESTAMP | TIMESTAMP |
| TINYINT | BIGINT |
| VARBINARY | VARBINARY |
| VARCHAR | VARCHAR |

Cache Data Type Mapping

TDV supports caching to a limited number of data source types, and the data type mapping for caching is specific to each type of data source.

The native data types supported for storing cache data are described in the following tables:

- [DB2 Cache Mapping, page 486](#)
- [DB2-on-z/OS Cache Mapping, page 488](#)
- [File Cache Mapping, page 489](#)
- [Greenplum Cache Mapping, page 492](#)
- [HSQLDB Cache Mapping, page 493](#)
- [Informix Cache Mapping, page 495](#)
- [Microsoft Access Cache Mapping, page 496](#)
- [Microsoft SQL Server Cache Mapping, page 498](#)
- [MySQL Cache Mapping, page 500](#)
- [Netezza Cache Mapping, page 502](#)
- [Oracle Cache Mapping, page 504](#)
- [PostgreSQL Cache Mapping, page 505](#)
- [Redshift Cache Mapping, page 507](#)
- [SAP HANA Cache Mapping, page 509](#)
- [Sybase ASE Cache Mapping, page 511](#)
- [Sybase IQ Cache Mapping, page 513](#)

- [Teradata Cache Mapping, page 514](#)
- [Vertica Cache Mapping, page 516](#)

The columns in the data type mapping tables provide this information:

- **Data Type**—Data types from a view or procedure output parameter.
- **Native Type or Preferred Native Type**—The data type that is suggested in the DDL when using the feature to create or recreate tables from Studio.
- **Other Allowed Native Types**—Other data types in the database (if any) that can be used as alternatives to the preferred type. A plus-sign (+) after a number means “or greater.”

DB2 Cache Mapping

The data type mappings for caches stored on DB2 are as follows.

| Data Type | Preferred Native Type | Other Allowed Native Types |
|--------------|-----------------------------------|---|
| BIGINT | BIGINT | DECIMAL(19+,0), larger INTEGER types, VARCHAR(20+) |
| BINARY(n) | BLOB | |
| BIT | SMALLINT | DECIMAL(1+,0), larger INTEGER types |
| BLOB | BLOB | |
| BOOLEAN | SMALLINT | INTEGER, BIGINT |
| CHAR(n) | CHAR(n); CLOB [if n > 254] | CHAR(n+), GRAPHIC(n+), VARCHAR(n+), VARGRAPHIC(n+), CLOB |
| CLOB | CLOB | LONG VARGRAPHIC |
| DATE | DATE | VARCHAR(10+) |
| DECIMAL(p,s) | DECIMAL(p,s); CLOB [if p > 31] | DECIMAL(p+,s+), VARCHAR(p+3+), VARGRAPHIC(p+3+), CLOB, LONG VARGRAPHIC |
| DOUBLE | DOUBLE | VARCHAR(24+) |

| Data Type | Preferred Native Type | Other Allowed Native Types |
|------------------------------|-----------------------------------|--|
| FLOAT | DOUBLE | VARCHAR(24+) |
| INTEGER | INTEGER | DECIMAL(10+,0), larger INTEGER types, VARCHAR(20+) |
| INTERVAL DAY | VARCHAR(30) | |
| INTERVAL DAY TO HOUR | VARCHAR(30) | |
| INTERVAL DAY TO MINUTE | VARCHAR(30) | |
| INTERVAL DAY TO SECOND | VARCHAR(30) | |
| INTERVAL HOUR | VARCHAR(30) | |
| INTERVAL HOUR TO MINUTE | VARCHAR(30) | |
| INTERVAL HOUR TO SECOND | VARCHAR(30) | |
| INTERVAL MINUTE | VARCHAR(30) | |
| INTERVAL MINUTE TO SECOND | VARCHAR(30) | |
| INTERVAL MONTH | VARCHAR(9) | |
| INTERVAL SECOND | VARCHAR(30) | |
| INTERVAL YEAR | VARCHAR(9) | |
| INTERVAL YEAR TO MONTH | VARCHAR(12) | |
| NUMERIC(p,s) | DECIMAL(p,s); CLOB [if p > 31] | DECIMAL(p+,s+), VARCHAR(p+3+), GRAPHIC(p+3+), CLOB |
| OTHER | [cannot be cached] | |
| REAL | REAL | |

| Data Type | Preferred Native Type | Other Allowed Native Types |
|--------------|----------------------------------|---|
| SMALLINT | SMALLINT | DECIMAL(5+,0), larger INTEGER types, VARCHAR(20+) |
| TIME | TIME | VARCHAR(15+) |
| TIMESTAMP | TIMESTAMP | VARCHAR(26+) |
| TINYINT | SMALLINT | DECIMAL(3+,0), larger INTEGER types, VARCHAR(20+) |
| VARBINARY(n) | BLOB | |
| VARCHAR(n) | VARCHAR(n); CLOB [if n > 254] | VARCHAR(n+), VARGRAPHIC(n+), CLOB, LONG VARGRAPHIC |
| XML | CLOB | VARCHAR(*) [truncates data if column is too small], VARGRAPHIC(*), LONG VARGRAPHIC |

DB2-on-z/OS Cache Mapping

The data type mappings for caches stored on DB2 (z/OS platform) are as follows.

| Data Type | Native Type |
|-----------|-------------------------------|
| BIGINT | BIGINT |
| BINARY | BLOB |
| BIT | SMALLINT |
| BLOB | BLOB |
| BOOLEAN | SMALLINT |
| CHAR | CHAR CLOB (if > 254 bytes) |
| CLOB | CLOB |
| DATE | DATE |

| Data Type | Native Type |
|-----------|------------------------------------|
| DECIMAL | DECIMAL CLOB (if precision >31) |
| DOUBLE | DOUBLE |
| FLOAT | DOUBLE |
| INTEGER | INTEGER |
| NUMERIC | DECIMAL CLOB (if precision >31) |
| REAL | REAL |
| SMALLINT | SMALLINT |
| TIME | TIME |
| TIMESTAMP | TIMESTAMP |
| TINYINT | SMALLINT |
| VARBINARY | BLOB |
| VARCHAR | VARCHAR CLOB (if length >255) |
| XML | CLOB |

File Cache Mapping

The data type mappings for caches stored in files are shown in the table. Any other data types cannot be cached.

| Data Type | Preferred Native Type | Other Allowed Native Types |
|-----------|-----------------------|--|
| BIGINT | BIGINT | DECIMAL(19+,0), larger INTEGER types, VARCHAR(20+) |
| BINARY(n) | BINARY(n) BLOB | BINARY(n+), BLOB |
| BIT | BIT | DECIMAL(1+,0), larger INTEGER types |

| Data Type | Preferred Native Type | Other Allowed Native Types |
|------------------------------|-----------------------|--|
| BLOB | BLOB | |
| CHAR(n) | CHAR(n) CLOB | CHAR(n+), CLOB |
| CLOB | CLOB | |
| DATE | DATE | VARCHAR(10+) |
| DECIMAL(p,s) | DECIMAL(p,s) | DECIMAL(p+,s+), VARCHAR(p+3+), CLOB, INTEGER types with enough resolution |
| DOUBLE | DOUBLE | VARCHAR(24+) |
| FLOAT | FLOAT | DOUBLE |
| INTEGER | INTEGER | DECIMAL(10+,0), larger INTEGER types, VARCHAR(20+) |
| INTERVAL DAY | VARCHAR(30) | |
| INTERVAL DAY TO HOUR | VARCHAR(30) | |
| INTERVAL DAY TO MINUTE | VARCHAR(30) | |
| INTERVAL DAY TO SECOND | VARCHAR(30) | |
| INTERVAL HOUR | VARCHAR(30) | |
| INTERVAL HOUR TO MINUTE | VARCHAR(30) | |
| INTERVAL HOUR TO SECOND | VARCHAR(30) | |
| INTERVAL MINUTE | VARCHAR(30) | |
| INTERVAL MINUTE TO SECOND | VARCHAR(30) | |

| Data Type | Preferred Native Type | Other Allowed Native Types |
|--|-----------------------|--|
| INTERVAL MONTH | VARCHAR(9) | |
| INTERVAL SECOND | VARCHAR(30) | |
| INTERVAL YEAR | VARCHAR(9) | |
| INTERVAL YEAR TO MONTH | VARCHAR(12) | |
| NUMERIC(p,s) | NUMERIC(p,s) | DECIMAL(p+,s+), VARCHAR(p+3+), CLOB, INTEGER types with enough resolution |
| OTHER | [cannot be cached] | |
| REAL | REAL | |
| SMALLINT | SMALLINT | DECIMAL(5+0), larger INTEGER types, VARCHAR(20+) |
| TIME | TIME | VARCHAR(15+) |
| TIMESTAMP | TIMESTAMP | |
| TINYINT | TINYINT | DECIMAL(3+,0), larger INTEGER types, VARCHAR(20+) |
| VARBINARY(n) | VARBINARY(n) BLOB | VARBINARY(n+), BLOB |
| VARCHAR(n) What is PROMOTE threshold for this data type in file caches? | VARCHAR(n) CLOB | VARCHAR(n+), CLOB |
| XML | CLOB | VARCHAR(*) Truncates data if column is too small. |

Greenplum Cache Mapping

The data type mappings for caches stored on Greenplum areas follows.

| Data Type | Native Type |
|-------------------------|------------------|
| BIGINT | BIGINT |
| BINARY(n) | BYTEA OID |
| BIT | SMALLINT |
| BLOB | OID |
| BOOLEAN | BOOLEAN |
| CHAR(n) | CHAR(n) TEXT |
| CLOB | TEXT |
| DATE | DATE |
| DECIMAL | DECIMAL(p,s) |
| DOUBLE | DOUBLE PRECISION |
| FLOAT | REAL |
| INTEGER | INTEGER |
| INTERVAL DAY | VARCHAR(30) |
| INTERVAL DAY TO HOUR | VARCHAR(30) |
| INTERVAL DAY TO MINUTE | VARCHAR(30) |
| INTERVAL DAY TO SECOND | VARCHAR(30) |
| INTERVAL HOUR | VARCHAR(30) |
| INTERVAL HOUR TO MINUTE | VARCHAR(30) |
| INTERVAL HOUR TO SECOND | VARCHAR(30) |
| INTERVAL MINUTE | VARCHAR(30) |

| Data Type | Native Type |
|---------------------------|--------------------|
| INTERVAL MINUTE TO SECOND | VARCHAR(30) |
| INTERVAL MONTH | VARCHAR(9) |
| INTERVAL SECOND | VARCHAR(30) |
| INTERVAL YEAR | VARCHAR(9) |
| INTERVAL YEAR TO MONTH | VARCHAR(12) |
| NUMERIC | NUMERIC(p,s) |
| REAL | REAL |
| SMALLINT | SMALLINT |
| TIME | TIME |
| TIMESTAMP | VARCHAR(26) |
| TINYINT | SMALLINT |
| VARBINARY(n) | BYTEA OID |
| VARCHAR(n) | VARCHAR(n) TEXT |
| XML | XML |

HSQldb Cache Mapping

The data type mappings for caches stored on HSQldb areas follows.

| Data Type | Native Type |
|--------------|--------------|
| BIGINT | BIGINT |
| BIT | SMALLINT |
| BOOLEAN | BOOLEAN |
| CHAR | CHAR(length) |
| CHAR_PROMOTE | CLOB |

| Data Type | Native Type |
|---------------------------|---------------------------|
| CLOB | CLOB |
| DATE | DATE |
| DECIMAL | NUMERIC(precision, scale) |
| DECIMAL_PROMOTE | BIGDECIMAL |
| DOUBLE | DOUBLE PRECISION |
| FLOAT | REAL |
| INTEGER | INTEGER |
| INTERVAL_DAY | INTERVAL DAY |
| INTERVAL_DAY_TO_HOUR | INTERVAL DAY TO HOUR |
| INTERVAL_DAY_TO_MINUTE | INTERVAL DAY TO MINUTE |
| INTERVAL_DAY_TO_SECOND | INTERVAL DAY TO SECOND |
| INTERVAL_HOUR | INTERVAL HOUR |
| INTERVAL_HOUR_TO_MINUTE | INTERVAL HOUR TO MINUTE |
| INTERVAL_HOUR_TO_SECOND | INTERVAL HOUR TO SECOND |
| INTERVAL_MINUTE | INTERVAL MINUTE |
| INTERVAL_MINUTE_TO_SECOND | INTERVAL MINUTE TO SECOND |
| INTERVAL_MONTH | INTERVAL MONTH |
| INTERVAL_SECOND | INTERVAL SECOND |
| INTERVAL_YEAR | INTERVAL YEAR |
| INTERVAL_YEAR_TO_MONTH | INTERVAL YEAR TO MONTH |
| NUMERIC | NUMERIC(precision, scale) |
| NUMERIC_PROMOTE | BIGDECIMAL |
| REAL | REAL |

| Data Type | Native Type |
|-----------------|-----------------|
| SMALLINT | SMALLINT |
| TIME | TIME |
| TIMESTAMP | TIMESTAMP |
| TINYINT | SMALLINT |
| VARCHAR | VARCHAR(length) |
| VARCHAR_PROMOTE | CLOB |
| XML | LONGVARCHAR |

Informix Cache Mapping

Informix Access does not have a TIME data type.

The data type mappings for caches stored on Informix are as follows.

| Data Type | Native Type |
|-----------|------------------|
| BIGINT | INT8 |
| BINARY | BYTE(n) BLOB |
| BIT | BOOLEAN |
| BLOB | BLOB |
| CHAR | CHAR(n) TEXT |
| CLOB | TEXT |
| DATE | DATE |
| DECIMAL | DECIMAL(p,s) |
| DOUBLE | DOUBLE PRECISION |
| FLOAT | REAL |
| INTEGER | INTEGER |

| Data Type | Native Type |
|---------------------------|--------------------|
| INTERVAL DAY | VARCHAR(30) |
| INTERVAL DAY TO HOUR | VARCHAR(30) |
| INTERVAL DAY TO MINUTE | VARCHAR(30) |
| INTERVAL DAY TO SECOND | VARCHAR(30) |
| INTERVAL HOUR | VARCHAR(30) |
| INTERVAL HOUR TO MINUTE | VARCHAR(30) |
| INTERVAL HOUR TO SECOND | VARCHAR(30) |
| INTERVAL MINUTE | VARCHAR(30) |
| INTERVAL MINUTE TO SECOND | VARCHAR(30) |
| INTERVAL MONTH | VARCHAR(9) |
| INTERVAL SECOND | VARCHAR(30) |
| INTERVAL YEAR | VARCHAR(9) |
| INTERVAL YEAR TO MONTH | VARCHAR(12) |
| NUMERIC | NUMERIC(p,s) |
| REAL | REAL |
| SMALLINT | SMALLINT |
| TIMESTAMP | DATETIME |
| TINYINT | SMALLINT |
| VARBINARY(n) | BYTE(n) BLOB |
| VARCHAR(n) | VARCHAR(n) TEXT |

Microsoft Access Cache Mapping

Microsoft Access does not have a TIME data type.

The data type mappings for caches stored on Microsoft Access are as follows.

| Data Type | Native Type |
|---------------------------|---------------|
| BIGINT | DECIMAL(20,0) |
| BINARY | LONGBINARY |
| BIT | BIT |
| BLOB | LONGBINARY |
| CHAR | VARCHAR(n) |
| CLOB | LONGCHAR |
| DATE | DATETIME |
| DECIMAL | DECIMAL(p,s) |
| DOUBLE | DOUBLE |
| FLOAT | FLOAT |
| INTEGER | INTEGER |
| INTERVAL DAY | VARCHAR(30) |
| INTERVAL DAY TO HOUR | VARCHAR(30) |
| INTERVAL DAY TO MINUTE | VARCHAR(30) |
| INTERVAL DAY TO SECOND | VARCHAR(30) |
| INTERVAL HOUR | VARCHAR(30) |
| INTERVAL HOUR TO MINUTE | VARCHAR(30) |
| INTERVAL HOUR TO SECOND | VARCHAR(30) |
| INTERVAL MINUTE | VARCHAR(30) |
| INTERVAL MINUTE TO SECOND | VARCHAR(30) |
| INTERVAL MONTH | VARCHAR(9) |
| INTERVAL SECOND | VARCHAR(30) |

| Data Type | Native Type |
|------------------------|--------------|
| INTERVAL YEAR | VARCHAR(9) |
| INTERVAL YEAR TO MONTH | VARCHAR(12) |
| NUMERIC | DECIMAL(p,s) |
| REAL | FLOAT |
| SMALLINT | SMALLINT |
| TIMESTAMP | DATETIME |
| TINYINT | BYTE |
| VARBINARY | LONGBINARY |
| VARCHAR | VARCHAR(n) |

Microsoft SQL Server Cache Mapping

This section discusses the data type mappings and restrictions for caches stored on Microsoft SQL Server 2000, 2005, or 2008. Overrides for the 2008 version are indicated in square brackets.

- SQL Server’s page size limits the number of bytes that can be stored directly in a column—so executing DDL causes an error if the resulting table requires a row size greater than this limit. The solution is to either raise the page size for the database, or to use indirect storage types such as TEXT and IMAGE. TDV chooses TEXT and IMAGE types if a value requires more than 255 bytes of storage for this reason, although SQL Server does allow VARCHAR and VARBINARY up to 8,000 bytes. Hand-tuning of the data types used in a table can improve storage efficiency.
- Microsoft SQL Server TINYINT has a range 0 to 255, and TDV TINYINT is -128 to 127, so these types are not compatible.
- DATETIME has only 3.33ms accuracy, so rounding error may occur.

| Data Type | Preferred Native Type | Other Allowed Native Types |
|-----------|-----------------------|---|
| BIGINT | BIGINT | DECIMAL(19+,0), larger INTEGER types, VARCHAR(20+), NVARCHAR(20+) |

| Data Type | Preferred Native Type | Other Allowed Native Types |
|--------------|-----------------------------------|---|
| BINARY(n) | BINARY(n); IMAGE [if n > 255] | BINARY(n+), IMAGE |
| BIT | BIT | DECIMAL(1+,0), larger INTEGER types |
| BLOB | IMAGE | |
| BOOLEAN | BIT | TINYINT, SMALLINT, INTEGER, BIGINT |
| CHAR(n) | CHAR(n); TEXT [if p > 38] | CHAR(n+), NCHAR(n+), VARCHAR(n+), NVARCHAR(n+), TEXT, NTEXT |
| CLOB | TEXT | NTEXT |
| DATE | DATE [2008] VARCHAR(10) | VARCHAR(10+) |
| DECIMAL(p,s) | DECIMAL(p,s); TEXT [if p > 38] | DECIMAL(p+,s+), VARCHAR(p+3+), NVARCHAR(n+), TEXT, NTEXT |
| DOUBLE | FLOAT | VARCHAR(24+) |
| FLOAT | REAL | FLOAT, VARCHAR(24+) |
| INTEGER | INTEGER | DECIMAL(10+,0), larger INTEGER types, VARCHAR(20+), NVARCHAR(20+) |
| NUMERIC(p,s) | DECIMAL(p,s); TEXT [if p > 38] | DECIMAL(p+,s+), VARCHAR(p+3+), NVARCHAR(p+3+), TEXT, NTEXT |
| OTHER | [cannot be cached] | |
| SMALLINT | SMALLINT | DECIMAL(5+,0), larger INTEGER types, VARCHAR(20+), NVARCHAR(20+) |
| TIME | TIME [2008] VARCHAR(15) | VARCHAR(15+) |

| Data Type | Preferred Native Type | Other Allowed Native Types |
|--------------|-------------------------------------|--|
| TIMESTAMP | DATETIME2 [2008] DATETIME | |
| TINYINT | SMALLINT | DECIMAL(3+,0), larger INTEGER types, VARCHAR(20+), NVARCHAR(20+) |
| VARBINARY(n) | VARBINARY(n); IMAGE [if n > 255] | VARBINARY(n+), IMAGE |
| VARCHAR(n) | VARCHAR(n); TEXT [if n > 255] | VARCHAR(n+), NVARCHAR(n+), TEXT, NTEXT |
| XML | TEXT | VARCHAR(*) [Truncates data if column is too small], TEXT |

MySQL Cache Mapping

This section discusses the data type mappings and restrictions for caches stored on MySQL.

- MySQL removes trailing spaces from strings stored in a VARCHAR column and trailing 0x20 bytes from a VARBINARY column.
- MySQL truncates millisecond data from TIME, DATETIME, and TIMESTAMP columns.
- MySQL changes any NULL stored in a TIMESTAMP column into the current date. Use DATETIME to preserve NULL values.
- TDV creates tables using the UTF8 character set to handle international characters properly. You can create the tables using other character sets based on your performance and character set needs.
- Small variations in the least significant digits may be encountered when storing FLOAT and DOUBLE values due to the way the driver handles and database stores such data.

The following table shows the mapping from TDV data types to native types.

| Data Type | Preferred Native Type | Other Allowed Native Types |
|-----------|-----------------------|---|
| BIGINT | BIGINT | DECIMAL(19+, 0), larger INTEGER types, VARCHAR(20+) |

| Data Type | Preferred Native Type | Other Allowed Native Types |
|--------------|-----------------------------------|---|
| BINARY(n) | BLOB; LONGBLOB [if n > 255] | TINYBLOB, BLOB, MEDIUMBLOB, LONGBLOB |
| BIT | BIT | DECIMAL(1+, 0), larger INTEGER types |
| BLOB | LONGBLOB | |
| BOOLEAN | BIT | BIT, BOOL |
| CHAR(n) | CHAR(n); LONGTEXT [if n > 255] | CHAR(n+), TINYTEXT, TEXT, MEDIUMTEXT, LONGTEXT |
| CLOB | CLOB | |
| DATE | DATE | VARCHAR(10+) |
| DECIMAL(p,s) | DECIMAL(p,s); TEXT [if p > 30] | DECIMAL(p+,s+), VARCHAR(p+3+), TINYTEXT, MEDIUMTEXT, LONGTEXT, INTEGER types with enough resolution |
| DOUBLE | DOUBLE | VARCHAR(24+) |
| FLOAT | FLOAT | VARCHAR(24+) |
| INTEGER | INTEGER | DECIMAL(10+, 0), larger INTEGER types, VARCHAR(20+) |
| NUMERIC(p,s) | NUMERIC(p,s); TEXT [if p > 30] | DECIMAL(p+,s+), VARCHAR(p+3+), TINYTEXT, MEDIUMTEXT, LONGTEXT, INTEGER types with enough resolution |
| SMALLINT | SMALLINT | DECIMAL(5+, 0), larger INTEGER types, VARCHAR(20+) |
| TIME | TIME | VARCHAR(15+) |
| TIMESTAMP | DATETIME | TIMESTAMP |
| TINYINT | TINYINT | DECIMAL(3+, 0), larger INTEGER types, VARCHAR(20+) |

| Data Type | Preferred Native Type | Other Allowed Native Types |
|--------------|-----------------------------------|--|
| VARBINARY(n) | BLOB; LONGBLOB [if n > 255] | TINYBLOB, BLOB, MEDIUMBLOB, LONGBLOB |
| VARCHAR(n) | VARCHAR(n); LONGTEXT [if n > 255] | VARCHAR(n+), TINYTEXT, TEXT, MEDIUMTEXT, LONGTEXT |
| XML | LONGTEXT | VARCHAR(*), TINYINT, TEXT, MEDIUMTEXT [Truncates data if column too small] |

Netezza Cache Mapping

This section discusses the data type mappings for caches stored on Netezza.

Netezza data types have these characteristics:

- IN predicate (with multiple values) with a subquery. It does not support multiple values on the right hand side. For example, Pushable (x,y) IN (select a,b from foo) : Not Pushable - will be processed within TDV - (k, j) IN ((a, b), (c, d), (e, f))

Overrides for versions 5.0 and 6.0 are indicated in square brackets.

| Data Type | Native Type |
|-----------|--------------------------------------|
| BIGINT | BIGINT, INT8 [not 5.0, 6.0] |
| BIT | BOOLEAN |
| BOOL | BOOLEAN |
| BOOLEAN | BOOLEAN |
| CHAR | CHAR(n), CHAR [not 5.0, 6.0] |
| DATE | DATE |
| DECIMAL | NUMERIC(p,s), DECIMAL [not 5.0, 6.0] |
| DOUBLE | DOUBLE PRECISION |
| FLOAT | FLOAT |
| INTEGER | INT, INT4, INTEGER |

| Data Type | Native Type |
|---------------------------|-------------------------------|
| INTERVAL | VARCHAR |
| INTERVAL DAY | VARCHAR(30) |
| INTERVAL DAY TO HOUR | VARCHAR(30) |
| INTERVAL DAY TO MINUTE | VARCHAR(30) |
| INTERVAL DAY TO SECOND | VARCHAR(30) |
| INTERVAL HOUR | VARCHAR(30) |
| INTERVAL HOUR TO MINUTE | VARCHAR(30) |
| INTERVAL HOUR TO SECOND | VARCHAR(30) |
| INTERVAL MINUTE | VARCHAR(30) |
| INTERVAL MINUTE TO SECOND | VARCHAR(30) |
| INTERVAL MONTH | VARCHAR(9) |
| INTERVAL SECOND | VARCHAR(30) |
| INTERVAL YEAR | VARCHAR(9) |
| INTERVAL YEAR TO MONTH | VARCHAR(12) |
| NUMERIC | NUMERIC(p,s) |
| REAL | REAL |
| SMALLINT | SMALLINT, INT2 |
| TIME | TIME, TIMEZ |
| TIMESTAMP | TIMESTAMP |
| TIMETZ(n) | TIMETZ [5.0, 6.0] |
| TINYINT | SMALLINT, INT1 [not 5.0, 6.0] |
| VARCHAR | VARCHAR(n) |

Oracle Cache Mapping

This section discusses the data type mappings and restrictions for caches stored on Oracle.

Oracle changes any empty string stored in a VARCHAR2 or NVARCHAR2 column to NULL. This can alter empty string data stored in such columns.

FLOAT columns have a maximum of 126 digits, equivalent to a floating-point number with exponent E125. TDV FLOAT values have a maximum of E38 and DOUBLE values have a maximum of E308. This is why VARCHAR is used to store TDV DOUBLE values by default. However, you can use the FLOAT type if your values fit within that range.

| Data Type | Preferred Data Type | Other Allowed Native Types |
|--------------|----------------------------------|--|
| BIGINT | NUMBER(19, 0) | NUMBER(19+, 0), VARCHAR(20+), NVARCHAR(20+) |
| BINARY(n) | RAW(n); BLOB [if n > 255] | RAW(n+), BLOB |
| BIT | NUMBER(1, 0) | NUMBER(1+, 0) |
| BLOB | BLOB | |
| BOOLEAN | NUMBER(1,0) | NUMBER(1+,0) |
| CHAR(n) | CHAR(n); CLOB [if n > 2000] | CHAR(n+), VARCHAR2(n+), NVARCHAR2(n+), CLOB |
| CLOB | CLOB | |
| DATE | VARCHAR2(10) | VARCHAR2(10+), NVARCHAR2(10+) |
| DECIMAL(p,s) | NUMBER(p,s); CLOB [if p > 38] | NUMBER(p+,s+), VARCHAR2(p+ 3+), NVARCHAR2(p+3+), CLOB |
| DOUBLE | VARCHAR(24) | VARCHAR(24+), FLOAT, BINARY DOUBLE |
| FLOAT | FLOAT | VARCHAR(24+), FLOAT, BINARY FLOAT, BINARY DOUBLE |

| Data Type | Preferred Data Type | Other Allowed Native Types |
|--------------|-----------------------------------|--|
| INTEGER | NUMBER(10, 0) | NUMBER(10+, 0), VARCHAR(20+), NVARCHAR(20+) |
| NUMERIC(p,s) | NUMBER(p,s); CLOB [if p > 38] | NUMBER(p+,s+), VARCHAR2(p+ 3+), NVARCHAR2(p+3+), CLOB |
| OTHER | [cannot be cached] | |
| SMALLINT | NUMBER(5, 0) | NUMBER(5+, 0), VARCHAR(20+), NVARCHAR(20+) |
| TIME | VARCHAR2(15) | VARCHAR2(15+), NVARCHAR2(15+) |
| TIMESTAMP | TIMESTAMP(9) [9i, 10g] | |
| TINYINT | NUMBER(3, 0) | NUMBER(3+, 0), VARCHAR(20+), NVARCHAR(20+) |
| VARBINARY(n) | RAW(n); BLOB [if n > 255] | RAW(n+), BLOB |
| VARCHAR(n) | VARCHAR(n); CLOB [if n > 4000] | VARCHAR2(n+), NVARCHAR2(n+), CLOB |
| XML | CLOB | VARCHAR(*), NVARCHAR2(*) [Truncates data if column is too small] |

PostgreSQL Cache Mapping

The data type mappings for caches stored on PostgreSQL are as follows.

| Data Type | Native Type |
|-----------|----------------------------|
| BIGINT | BIGINT |
| BINARY | BYTEA [regardless of size] |
| BIT | BOOLEAN |
| BLOB | BYTEA |

| Data Type | Native Type |
|---------------------------|----------------------|
| BOOLEAN | BOOLEAN |
| CHAR | CHAR(n) TEXT |
| CLOB | TEXT |
| DATE | DATE |
| DECIMAL | DECIMAL TEXT |
| DOUBLE | DOUBLE PRECISION |
| FLOAT | REAL |
| INTEGER | INTEGER |
| INTERVAL DAY | VARCHAR(30) |
| INTERVAL DAY TO HOUR | VARCHAR(30) |
| INTERVAL DAY TO MINUTE | VARCHAR(30) |
| INTERVAL DAY TO SECOND | VARCHAR(30) |
| INTERVAL HOUR | VARCHAR(30) |
| INTERVAL HOUR TO MINUTE | VARCHAR(30) |
| INTERVAL HOUR TO SECOND | VARCHAR(30) |
| INTERVAL MINUTE | VARCHAR(30) |
| INTERVAL MINUTE TO SECOND | VARCHAR(30) |
| INTERVAL MONTH | VARCHAR(9) |
| INTERVAL SECOND | VARCHAR(30) |
| INTERVAL YEAR | VARCHAR(9) |
| INTERVAL YEAR TO MONTH | VARCHAR(12) |
| NUMERIC | NUMERIC(p,s) TEXT |

| Data Type | Native Type |
|--------------|----------------------------|
| REAL | REAL |
| SMALLINT | SMALLINT |
| TIME | TIME |
| TIMESTAMP | TIMESTAMP |
| TINYINT | SMALLINT |
| VARBINARY(n) | BYTEA [regardless of size] |
| VARCHAR(n) | VARCHAR(n) TEXT |
| XML | XML |

Redshift Cache Mapping

This section discusses the data type mappings for caches stored on Redshift.

| Data Type | Native Type |
|----------------|---------------|
| BIGINT | BIGINT |
| BINARY | VARCHAR |
| BINARY_PROMOTE | VARCHAR |
| BIT | SMALLINT |
| BLOB | VARCHAR |
| BOOLEAN | BOOLEAN |
| CHAR | CHAR(&L) |
| CHAR_PROMOTE | TEXT |
| CLOB | TEXT |
| DATE | DATE |
| DECIMAL | NUMERIC(p, s) |

| Data Type | Native Type |
|---------------------------|------------------|
| DECIMAL_PROMOTE | TEXT |
| DOUBLE | DOUBLE PRECISION |
| FLOAT | REAL |
| INTEGER | INTEGER |
| INTERVAL_DAY | VARCHAR |
| INTERVAL_DAY_TO_HOUR | VARCHAR |
| INTERVAL_DAY_TO_MINUTE | VARCHAR |
| INTERVAL_DAY_TO_SECOND | VARCHAR |
| INTERVAL_HOUR | VARCHAR |
| INTERVAL_HOUR_TO_MINUTE | VARCHAR |
| INTERVAL_HOUR_TO_SECOND | VARCHAR |
| INTERVAL_MINUTE | VARCHAR |
| INTERVAL_MINUTE_TO_SECOND | VARCHAR |
| INTERVAL_MONTH | VARCHAR |
| INTERVAL_SECOND | VARCHAR |
| INTERVAL_YEAR | VARCHAR |
| INTERVAL_YEAR_TO_MONTH | VARCHAR |
| NUMERIC | NUMERIC(p, s) |
| NUMERIC_PROMOTE | TEXT |
| REAL | REAL |
| SMALLINT | SMALLINT |
| TIME | TIME |
| TIMESTAMP | TIMESTAMP |

| Data Type | Native Type |
|-------------------|-----------------|
| TINYINT | SMALLINT |
| VARBINARY | VARCHAR |
| VARBINARY_PROMOTE | VARCHAR |
| VARCHAR | VARCHAR(length) |
| VARCHAR_PROMOTE | TEXT |
| XML | VARCHAR |

SAP HANA Cache Mapping

This section discusses the data type mappings for caches stored on SAP HANA.

| Data Type | Native Type |
|-----------------|--------------|
| DATE | DATE |
| TIME | TIME |
| TIMESTAMP | TIMESTAMP |
| BIT | TINYINT |
| TINYINT | SMALLINT |
| SMALLINT | SMALLINT |
| INTEGER | INTEGER |
| BIGINT | BIGINT |
| DECIMAL | DECIMAL(p,s) |
| DECIMAL_PROMOTE | CLOB |
| NUMERIC | DECIMAL(p,s) |
| NUMERIC_PROMOTE | CLOB |
| REAL | FLOAT(24) |
| FLOAT | DOUBLE |

| Data Type | Native Type |
|-------------------------------|-------------------|
| DOUBLE | DOUBLE |
| CHAR | NCHAR(length) |
| CHAR_PROMOTE | CLOB |
| VARCHAR | NVARCHAR(length) |
| VARCHAR_PROMOTE | CLOB |
| LONGVARCHAR | CLOB |
| BINARY | BINARY(length) |
| BINARY_PROMOTE | BLOB |
| VARBINARY | VARBINARY(length) |
| VARBINARY_PROMOTE | BLOB |
| BLOB | BLOB |
| CLOB | CLOB |
| XML | CLOB |
| BOOLEAN | TINYINT |
| INTERVAL_DAY | VARCHAR(30) |
| INTERVAL_DAY_TO_HOUR | VARCHAR(30) |
| INTERVAL_DAY_TO_MINUTE | VARCHAR(30) |
| INTERVAL_DAY_TO_SECOND | VARCHAR(30) |
| INTERVAL_HOUR | VARCHAR(30) |
| INTERVAL_HOUR_TO_MINUTE | VARCHAR(30) |
| INTERVAL_HOUR_TO_SECOND | VARCHAR(30) |
| INTERVAL_MINUTE | VARCHAR(30) |
| INTERVAL_MINUTE_TO_SECON D | VARCHAR(30) |

| Data Type | Native Type |
|------------------------|-------------|
| INTERVAL_SECOND | VARCHAR(30) |
| INTERVAL_YEAR | VARCHAR(9) |
| INTERVAL_YEAR_TO_MONTH | VARCHAR(12) |
| INTERVAL_MONTH | VARCHAR(9) |

Sybase ASE Cache Mapping

This section discusses the data type mappings and restrictions for caches stored on Sybase ASE.

Sybase ASE's page size limits the number of bytes that can be stored directly in a column. This means it is possible when executing DDL to get an error that the resulting table requires a row size greater than this limit. The solution is to either raise the page size for the database, or to use indirect storage types such as TEXT. For this reason, TDV chooses TEXT types if a value requires more than 255 bytes of storage, although Sybase ASE does allow VARCHAR and VARBINARY with larger size. Hand tuning of the data types used in a table can improve storage efficiency.

DATETIME has accuracy only to within 3.33ms, so some rounding error may occur.

| Data Type | Preferred Data Type | Other Allowed Native Types |
|-----------|----------------------------------|--|
| BIGINT | DECIMAL(19,0) | DECIMAL(19+,0) |
| BINARY(n) | BINARY(n); IMAGE [if n > 255] | BINARY(n+), IMAGE |
| BIT | BIT | DECIMAL(1+,0), larger INTEGER types |
| BLOB | IMAGE | |
| BOOLEAN | BIT | TINYINT, SMALLINT, INTEGER |
| CHAR(n) | CHAR(n); TEXT [if n > 255] | CHAR(n+), NCHAR(n+), VARCHAR(n+), NVARCHAR(n+), TEXT |
| CLOB | TEXT | NTEXT |

| Data Type | Preferred Data Type | Other Allowed Native Types |
|--------------|-------------------------------------|--|
| DATE | VARCHAR(10) | VARCHAR(10+) |
| DECIMAL(p,s) | DECIMAL(p,s); TEXT [if p > 38] | DECIMAL(p+,s+), VARCHAR(p+3+), NVARCHAR(p+3+), TEXT |
| DOUBLE | FLOAT | VARCHAR(24+) |
| FLOAT | REAL | FLOAT, VARCHAR(24+) |
| INTEGER | INT | DECIMAL(10+,0), VARCHAR(20+), NVARCHAR(20+) |
| NUMERIC(p,s) | DECIMAL(p,s); TEXT [if p > 38] | DECIMAL(p+,s+), VARCHAR(p+3+), NVARCHAR(p+3+), TEXT |
| OTHER | [cannot be cached] | |
| SMALLINT | SMALLINT | DECIMAL(5+,0), larger INTEGER types, VARCHAR(20+), NVARCHAR(20+) |
| TIME | VARCHAR(15) | VARCHAR(15+) |
| TIMESTAMP | VARCHAR(26) | DATETIME, VARCHAR(26+) |
| TINYINT | SMALLINT | DECIMAL(3+,0), larger INTEGER types, VARCHAR(20+), NVARCHAR(20+) |
| VARBINARY(n) | VARBINARY(n); IMAGE [if n > 255] | VARBINARY(n+), IMAGE |
| VARCHAR(n) | VARCHAR(n); TEXT [if n > 255] | VARCHAR(n+), NVARCHAR(n+), TEXT |
| XML | TEXT | VARCHAR(*) [Truncates data if column too small] |

Sybase IQ Cache Mapping

The data type mappings for caches stored on Sybase IQ are as follows.

| Data Type | Native Type |
|---------------------------|--------------------------------|
| BIGINT | DECIMAL(19,0) |
| BINARY | BINARY(n); IMAGE [if n > 255] |
| BIT | BIT |
| BLOB | IMAGE |
| BOOLEAN | BIT |
| CHAR | CHAR(n); TEXT [if n > 255] |
| CLOB | TEXT |
| DATE | DATE |
| DECIMALCHAR(n) | DECIMAL(p,s); TEXT [if p > 38] |
| DOUBLE | DOUBLE |
| FLOAT | REAL |
| INTEGER | INT |
| INTERVAL DAY | VARCHAR(30) |
| INTERVAL DAY TO HOUR | VARCHAR(30) |
| INTERVAL DAY TO MINUTE | VARCHAR(30) |
| INTERVAL DAY TO SECOND | VARCHAR(30) |
| INTERVAL HOUR | VARCHAR(30) |
| INTERVAL HOUR TO MINUTE | VARCHAR(30) |
| INTERVAL HOUR TO SECOND | VARCHAR(30) |
| INTERVAL MINUTE | VARCHAR(30) |
| INTERVAL MINUTE TO SECOND | VARCHAR(30) |

| Data Type | Native Type |
|------------------------|---|
| INTERVAL MONTH | VARCHAR(9) |
| INTERVAL SECOND | VARCHAR(30) |
| INTERVAL YEAR | VARCHAR(9) |
| INTERVAL YEAR TO MONTH | VARCHAR(12) |
| NUMERIC | NUMERIC(p,s); TEXT [if p > 38] |
| REAL | REAL |
| SMALLINT | SMALLINT |
| TIME | TIME |
| TIMESTAMP | TIMESTAMP For Sybase IQ 15.2 as a cache target, the year range is 0001 through 9999. |
| TINYINT | SMALLINT |
| VARBINARY | VARBINARY(n); IMAGE [if n > 255] |
| VARCHAR | VARCHAR(n); TEXT [if n > 255] |
| XML | TEXT |

Teradata Cache Mapping

This section discusses the data type mappings and restrictions for caches stored on Teradata. Data type overrides for versions 12 and 13 are indicated in square brackets. Data types not listed in the table cannot be cached.

| Data Type | Preferred Data Type | Other Allowed Native Types |
|-----------|----------------------------------|--|
| BIGINT | BIGINT [12, 13] CHAR(20) | DECIMAL(19+,0), VARCHAR(20+) |
| BINARY(n) | BYTE(n), BLOB [if n > 32,000] | BYTE(n+) |
| BIT | BYTEINT | DECIMAL(1+,0), larger INTEGER types |

| Data Type | Preferred Data Type | Other Allowed Native Types |
|--------------|-----------------------------------|---|
| BLOB | BLOB | |
| BOOLEAN | BYTEINT | SMALLINT, INTEGER |
| CHAR(n) | CHAR(n); CLOB [if n > 32,000] | CHAR(n+), GRAPHIC(n+), VARCHAR(n+), CLOB |
| CLOB | CLOB | Teradata 15 JDBC driver does not support CLOB column with NULL values when caching to Teradata 15. |
| DATE | DATE | VARCHAR(10+) |
| DECIMAL(p,s) | DECIMAL(p,s); CLOB [if p > 18] | DECIMAL(p+,s+), VARCHAR(p+3+), GRAPHIC(p+3+), CLOB |
| DOUBLE | FLOAT | VARCHAR(24+) |
| FLOAT | FLOAT | VARCHAR(24+) |
| INTEGER | INTEGER | DECIMAL(10+,0), VARCHAR(20+), GRAPHIC(20+), , VARGRAPHIC(20+) |
| NUMERIC(p,s) | DECIMAL(p,s); CLOB [if p > 18] | DECIMAL(p+,s+), VARCHAR(p+3+), GRAPHIC(p+3+), CLOB |
| OTHER | [cannot be cached] | |
| SMALLINT | SMALLINT | DECIMAL(5+,0), larger INTEGER types, VARCHAR(20+), VARGRAPHIC(20+) |
| TIME | VARCHAR(15) | VARCHAR(15+) |
| TIMESTAMP | TIMESTAMP | VARCHAR(26+) |
| TINYINT | BYTEINT | DECIMAL(3+,0), larger INTEGER types, VARCHAR(20+), VARGRAPHIC(20+) |

| Data Type | Preferred Data Type | Other Allowed Native Types |
|--------------|-------------------------------------|--|
| VARBINARY(n) | VARBYTE(n); BLOB [if n > 32,000] | VARBYTE(n+) |
| VARCHAR(n) | VARCHAR(n); CLOB [if n > 32,000] | VARCHAR(n+), VARGRAPHIC(n+) |
| XML | CLOB | VARCHAR(*) [Truncates data if column too small] |

Vertica Cache Mapping

Because of Vertica length limits, mapping of any data type (BINARY, CHAR, VARCHAR, BLOB, and so on) to Vertica cache with length greater than 65000 results in an error.

The data type mappings for caches stored on Vertica are as follows.

| Data Type | Native Type |
|-----------|------------------|
| BINARY(n) | BINARY(n) |
| BIGINT | INT8 |
| BIT | BOOLEAN |
| BLOB | VARBINARY(n) |
| BOOL | BOOLEAN |
| BOOLEAN | BOOLEAN |
| CHAR | CHAR(n) |
| CLOB | VARBINARY(n) |
| DATE | DATE |
| DECIMAL | DECIMAL(p,s) |
| DOUBLE | DOUBLE PRECISION |
| FLOAT | FLOAT |
| INTEGER | INTEGER |

| Data Type | Native Type |
|---------------------------|--------------|
| INTERVAL | VARCHAR |
| INTERVAL DAY | VARCHAR(30) |
| INTERVAL DAY TO HOUR | VARCHAR(30) |
| INTERVAL DAY TO MINUTE | VARCHAR(30) |
| INTERVAL DAY TO SECOND | VARCHAR(30) |
| INTERVAL HOUR | VARCHAR(30) |
| INTERVAL HOUR TO MINUTE | VARCHAR(30) |
| INTERVAL HOUR TO SECOND | VARCHAR(30) |
| INTERVAL MINUTE | VARCHAR(30) |
| INTERVAL MINUTE TO SECOND | VARCHAR(30) |
| INTERVAL MONTH | VARCHAR(9) |
| INTERVAL SECOND | VARCHAR(30) |
| INTERVAL YEAR | VARCHAR(9) |
| INTERVAL YEAR TO MONTH | VARCHAR(12) |
| LONG | BIGINT |
| NCLOB | VARBINARY(n) |
| NUMERIC | NUMERIC(p,s) |
| REAL | REAL |
| SMALLINT | SMALLINT |
| TIME | TIME, TIMEZ |
| TIMESTAMP | TIMESTAMP |
| TINYINT | TINYINT |
| VARBINARY(n) | VARBINARY(n) |

| Data Type | Native Type |
|-----------|-------------|
| VARCHAR | VARCHAR(n) |

Function Support for Data Sources

This topic lists all functions that can be pushed to each data source, by vendor. The first sections of this topic apply to every type of data source.

- [Pushing or Not Pushing Functions, page 520](#)
- [Function Support Issues when Combining Data Sources, page 520](#)
- [TDV Native Function Support, page 531](#)
- [DataDirect Mainframe Function Support, page 534](#)
- [DB2 Function Support, page 539](#)
- [DB2 Mainframe Function Support, page 546](#)
- [File Function Support, page 553](#)
- [Greenplum Function Support, page 556](#)
- [HBase Function Support, page 568](#)
- [HSQLDB Function Support, page 570](#)
- [Impala Function Support, page 575](#)
- [Informix Function Support, page 581](#)
- [JDBC Function Support, page 585](#)
- [JSON Function Support, page 585](#)
- [Microsoft Access Function Support, page 586](#)
- [Microsoft Access Function Support, page 586](#)
- [Microsoft Excel Function Support, page 590](#)
- [Microsoft SQL Server Function Support, page 590](#)
- [MySQL Function Support, page 597](#)
- [NeoView Function Support, page 602](#)
- [Netezza Function Support, page 605](#)
- [Oracle Function Support, page 618](#)
- [ParStream Function Support, page 628](#)
- [PostgreSQL Function Support, page 630](#)
- [Redshift Function Support, page 636](#)

- [SAP HANA Function Support, page 642](#)
- [Sybase Function Support, page 649](#)
- [Sybase IQ Function Support, page 653](#)
- [Teradata Function Support, page 658](#)
- [Vertica Function Support, page 664](#)
- [XML Function Support, page 678](#)

Pushing or Not Pushing Functions

A large number of SQL functions can be either executed within the TDV Server or pushed down to data sources for execution.

In general it is preferable to push function execution to the data source, for faster execution and reduced data transfer. However, for various reasons, such as query federation, it may be preferable not to push function execution to the data source. Query engine execution plans, or explicit SQL query options (described in [TDV Query Engine Options, page 245](#)), might force execution in the TDV Server rather than in the data source.

Refer to [TDV Support for SQL Functions, page 73](#), to see which functions can be executed in the TDV Server (that is, not pushed). TDV supports a wide variety of functions, although not every function available in every data source.

A few functions, such as DENSE_RANK and FIRST_VALUE, can be executed only in the data source. These are called “push-only” functions. [Function Support Summary, page 711](#), has a column that indicates which functions are push-only.

Because data sources implement many functions differently from each other and from TDV, results of execution might not be the same. The section [Function Support Issues when Combining Data Sources, page 520](#), discusses many of these differences.

Function Support Issues when Combining Data Sources

Data virtualization typically involves many data sources, each with its own collection of data types and functions and its own way of handling them. Besides this, queries and functions can be executed natively in the TDV Server. The number of combinations, therefore, is very large.

Several issues that might result from combining data sources are covered:

- [ASCII Function with Empty String Argument, page 521](#)
- [Case Sensitivity and Trailing Spaces, page 521](#)
- [Collating Sequence, page 521](#)
- [Data Precision, page 522](#)
- [Decimal Digit Limitation on Functions, page 523](#)
- [INSTR Function, page 523](#)
- [Interval Calculations, page 523](#)
- [Mapping of Native to TDV Data Types Across TDV Versions, page 524](#)
- [MERGE, page 524](#)
- [ORDER BY Clause, page 530](#)
- [SPACE Function, page 530](#)
- [SQL Server Sorting Order, page 530](#)
- [Time Functions, page 531](#)
- [Truncation vs. Rounding, page 531](#)

ASCII Function with Empty String Argument

When the ASCII function is applied to an empty string argument, what it returns varies for different data sources. For example, ASCII('') returns zero as implemented in PostgreSQL, Sybase and MySQL. It returns NULL as implemented in TDV, SQL Server, Oracle, and Informix.

Case Sensitivity and Trailing Spaces

Case sensitivity and treatment of trailing spaces can be controlled at the server, session, request, and query level, and might be the same or different for TDV and the data sources involved. For a detailed discussion of these settings, see the “TDV Configuration Options” topic of the *TDV Administration Guide*.

Collating Sequence

TDV uses binary collation and does not support changing the collation setting. So when the underlying data source’s collation setting is different, push and no-push query results might vary for queries that depend on collation—for example, a query that sorts on a column containing CHAR or VARCHAR data.

Data sources support different collating schemes (some support multiple collating schemes), and their defaults are not always the same as TDV. Furthermore, TDV cannot change data source collating schemes connection by connection or query by query, because most data sources do not allow that.

This difference in collation can cause unpredictable or incorrect results when columns contain special characters (% , - , and so on). Users should look for the following SQL constructs to make sure that their results are not affected by this difference:

- During JOINS, TDV picks SORT MERGE as the default join algorithm. When executing the SORT MERGE, TDV injects an ORDER BY clause on both sides. If one side of the join contains data source data, the sorting order might be different from what TDV expects, and so the MERGE process may produce incorrect results.

An option is to use {OPTION HASH} in SORT MERGE queries, forcing TDV to use a HASH algorithm instead of the SORT MERGE algorithm. Be aware, though, that the HASH algorithm uses more memory because the query engine needs to hash the smaller side and then stream the bigger side over it.

- In general, data sources may have different result when ORDER BY is pushed vs. executed within TDV.
- If a WHERE clause contains a predicate with special characters, results might differ between push and no-push.

A check box near the bottom of the Advanced tab for data sources lets you mark the data sources as Collation Sensitive. TDV does not use the SORT MERGE join algorithm if one of the data sources involved in the join is marked as collation sensitive.

In many situations you can specify a different collating scheme in the SQL (for example, using "COLLATE Latin1_General_BIN"), but this can interfere with indexing and thus affect performance.

Data Precision

FLOAT and REAL Precision

Many data sources treat FLOAT and REAL as single-precision, but TDV treats these data types as DOUBLE. Queries can therefore return different results (more or fewer significant digits) depending on whether they are pushed or not pushed.

INTEGER Precision

When an value of INTEGER type is divided by another value of INTEGER type, the result might be INTEGER or it might be some other SQL Standard exact numeric type with implementation-defined precision and scale. So, for example, dividing 10 by 3 might produce exactly 3, or it might produce 3.3333.

Decimal Digit Limitation on Functions

In TDV version 7.0.2 or later, add, subtract, multiply, divide, and modulo operators in functions follow SQL Server's behavior, which prevents precision/scale from exceeding 38 digits. But customers might need to wrap CASTs around columns in cached tables whose data types no longer match in such situations, so a configuration parameter has been made available to restore pre-7.0.2 behavior.

The name of the boolean configuration parameter is Decimal digit limitation in functions:

- When set to True (the default), add, subtract, multiply, divide, and modulo operators in functions prevent precision/scale from exceeding 38 digits.
- When set to False, add, subtract, multiply, divide, and modulo operators in functions allow precision/scale to exceeding 38 digits.

INSTR Function

If INSTR is executed in TDV, it returns NULL for INSTR('','C') and 0 for INSTR('','C').

Note: The difference is a space character. The C character is just an example.

When pushed to some databases, INSTR('','C') might return 0 instead of NULL.

Interval Calculations

The JDBC drivers of most data sources do not support mapping INTERVAL data types in the data source to INTERVAL data types in TDV. Instead, they are mapped to VARCHAR(13) in TDV. Because of this mapping, functions that involve comparison of numeric values (such as AVG, MAX, and MIN) can return incorrect results.

For example, '-99' is evaluated as greater than ' 99' (note the leading space character) for no-push interval calculations, because string comparisons consider ASCII collating order, in which space comes before minus-sign.

A workaround is to embed the CAST function. For example, when finding the maximum value in column c1, which is an interval, use:
`MAX(CAST(c1 AS INTERVAL MONTH TO DAY))`

Note: A notable exception is the PostgreSQL JDBC driver, which supports mapping INTERVAL data types to INTERVAL data types.

Mapping of Native to TDV Data Types Across TDV Versions

As of version 7.0, TDV supports the BOOLEAN data type. One result is that BOOL or BOOLEAN data types in data sources are now mapped to BOOLEAN in TDV rather than to CHAR or BIT.

Effects of this change can include:

- Existing caches (target tables) may become incompatible and may have to be re-created.
- Parts of queries that used to push completely may not push now.
- Some views and procedures may be impacted if, for example, they apply some function to the column introspected as a CHAR, and now that it is a BOOLEAN it is no longer a valid argument for that function (or operator, clause, and so on).
- If a column was used in a JOIN criterion or a WHERE predicate, the column might now require an explicit CAST to be compared to another value.

Possible remedies include:

- Re-create incompatible caches or target tables created in TDV versions prior to 7.0.
- Remap BOOLEAN back to CHAR or BIT in values.xml and reintrospect the data source.

MERGE

TDV uses SQL 2003/2008 MERGE syntax. TDV pushes MERGE if the data source supports it.

Federated merge is possible if the target table's database supports positioned updates, inserts and deletes in its JDBC driver.

MERGE and Data Sources

The following table lists data sources and their treatment when MERGE is involved.

| Data Source | Comments |
|----------------------------------|---|
| DB2 Versions 8 | <p>Supports ANSI MERGE 2003/2008. MERGE is pushed whenever possible. However, in the non-push (federated) case, the driver does not support some of the features required for full support.</p> <p>If the MERGE statement contains a WHEN NOT MATCHED THEN INSERT clause, the MERGE statement may fail. Newer versions of DB2 do not have this problem.</p> <p>The workaround is to change the MERGE statement so that it is completely pushed to DB2.</p> |
| DB2 Versions 9.5, 10.5, and z/OS | |
| MySQL | <p>Does not support MERGE. However, it does have REPLACE INTO and DUPLICATE KEY.</p> <p>For a TDV MERGE of MySQL data to succeed, the MySQL target table must have a primary key, and all columns in the primary key must be part of the MERGE.</p> <p>For a MERGE on tables from the same MySQL connection: if one ResultSet is modified, the driver closes the other ResultSet. The workaround is to create a copy of the data source so that you are using two different JDBC connections to the same data source.</p> |
| Netezza | Not possible to do a MERGE, because Netezza does not support updatable cursors. |
| Oracle | |
| SQL Server 2008, 2012 | |
| Sybase ASE | Version 15.7 is the first version of ASE to support MERGE. |
| Sybase IQ | <p>Versions up to and including 16 do NOT support MERGE.</p> <p>The JTDS driver for Sybase supports scrolling updatable result sets; the JConnect 7 driver does not.</p> |

| Data Source | Comments |
|-------------|---|
| Teradata | <p>Teradata 12 and 13 support SQL 2003 MERGE.</p> <p>Teradata 14 supports DELETE, but does not support search conditions in the WHEN clause.</p> <p>Federated MERGE may be possible under either of the following conditions:</p> <ul style="list-style-type: none">• The target table contains a column that is the only member of a unique index.• A column is a member of one or more unique indexes on the table, and all the columns of at least one unique index have been selected in the result set. |
| Vertica | Does not support federated MERGE because its driver does not support scrollable cursors. |
| Vertica 6.x | Supports ANSI SQL 2003 MERGE. |

MERGE Examples

This section includes a number of representative MERGE examples.

Example

This example tests the subquery IN clause.

```
PROC ( : !DSMAP)
PROCEDURE m_mixed(out x CURSOR)
BEGIN
  DECLARE guid VARCHAR(10) DEFAULT SUBSTRING('${ITEM_GUID}', 1, 10);
  DELETE FROM /users/composite/test/sources/oracle/DEV1/UPDATES ;
  INSERT INTO /users/composite/test/sources/oracle/DEV1/UPDATES (col_id,col_decimal, col_varchar)
VALUES(3,30,guid),(4,40,guid),(5,50,guid),(6,60,guid),(-1,-10,guid);

  MERGE INTO /users/composite/test/sources/oracle/DEV1/UPDATES
  USING (SELECT * FROM /shared/examples/ds_inventory/tutorial/inventorytransactions) inventorytransactions
  ON col_id = unitsreceived
  WHEN MATCHED AND guid = col_varchar and col_decimal IN (SELECT o10_id * 10 FROM
/users/composite/test/sources/oracle/DEV1/O10 WHERE o10_id IN (3,4)) THEN DELETE;
  OPEN x FOR SELECT col_id,col_char,col_tinyint,col_smallint,col_decimal FROM
/users/composite/test/sources/oracle/DEV1/UPDATES WHERE guid = col_varchar;
END
```

Example

This example tests Microsoft SQL Server.

```
PROC (SERIAL)
PROCEDURE m_pushed(out x CURSOR)
```

```

BEGIN
  DECLARE guid VARCHAR(10) DEFAULT SUBSTRING('${ITEM_GUID}', 1, 6) || '019';
  DELETE FROM /users/composite/test/sources/mssql_2k8/devstd/devstd/dbo/updates WHERE guid = c_varchar;
  INSERT INTO /users/composite/test/sources/mssql_2k8/devstd/devstd/dbo/updates (c_id, c_decimal, c_varchar)
  values(3, null, guid), (4, 40, guid);

  MERGE INTO /users/composite/test/sources/mssql_2k8/devstd/devstd/dbo/updates
  USING /users/composite/test/sources/mssql_2k8/devstd/devstd/dbo/s10
  ON c_id = S_id AND c_varchar = guid
  WHEN MATCHED AND c_decimal + 1 IS NOT NULL THEN UPDATE SET c_id = S_id + 10000 + c_id * 1000,
  c_char=S_char
  ;
  OPEN x FOR SELECT c_id, c_decimal, c_char FROM
  /users/composite/test/sources/mssql_2k8/devstd/devstd/dbo/updates WHERE c_varchar = guid;
END

```

Example

This example tests DB2.

```

PROC (DISABLED)
PROCEDURE m_mixed(out x CURSOR)
BEGIN
  DELETE FROM /users/composite/test/sources/"db2_9.5"/qa1_dev100_designbyexample/QA1/UPDATES;
  INSERT INTO /users/composite/test/sources/"db2_9.5"/qa1_dev100_designbyexample/QA1/UPDATES (c_id,
  c_decimal, c_varchar) values(3, null, '${ITEM_GUID}'), (4, 40, '${ITEM_GUID}');

  MERGE INTO /users/composite/test/sources/"db2_9.5"/qa1_dev100_designbyexample/QA1/UPDATES
  USING /users/composite/test/sources/mssql_2k8/devstd/devstd/dbo/s10
  ON c_id = S_id and c_varchar = '${ITEM_GUID}'
  WHEN NOT MATCHED THEN INSERT (c_id,c_char, c_varchar) VALUES (s_int, 'hey' || S_money,
  '${ITEM_GUID}');
  OPEN x FOR SELECT c_id, c_char FROM
  /users/composite/test/sources/"db2_9.5"/qa1_dev100_designbyexample/QA1/UPDATES WHERE c_varchar =
  '${ITEM_GUID}';
END

```

Example

In a MERGE statement, the same row of a table cannot be the target for combinations of UPDATE, DELETE and INSERT operations. This happens when a target row matches more than one source row. Refine the ON clause to ensure a target row matches at most one source row, or use the GROUP BY clause to group the source rows.

```

PROC
PROCEDURE m_pushed(out x CURSOR)
BEGIN
  DECLARE guid VARCHAR(10) DEFAULT SUBSTRING('${ITEM_GUID}', 1, 10);
  DELETE FROM /users/composite/test/sources/oracle/DEV1/UPDATES ;
  INSERT INTO /users/composite/test/sources/oracle/DEV1/UPDATES (col_id,col_decimal, col_varchar)
  VALUES(3,30, guid);

  MERGE INTO /users/composite/test/sources/oracle/DEV1/UPDATES
  USING (SELECT * FROM /shared/examples/ds_inventory/tutorial/inventorytransactions) inventorytransactions

```

```

ON col_id = purchaseorderid
WHEN MATCHED AND col_varchar = guid THEN UPDATE SET col_tinyint=productid;
END

```

Example

This example tests that DB2 does not allow a row to be deleted twice.

```

PROC
PROCEDURE m_error(out x CURSOR)
BEGIN
  DECLARE guid VARCHAR(10) DEFAULT SUBSTRING('${ITEM_GUID}', 1, 10);
  DELETE FROM /users/composite/test/sources/"db2_9.5"/qa1_dev100_designbyexample/QA1/UPDATES;
  INSERT INTO /users/composite/test/sources/"db2_9.5"/qa1_dev100_designbyexample/QA1/UPDATES (c_id,
c_decimal, c_varchar) values(1, null, guid);

  MERGE INTO /users/composite/test/sources/"db2_9.5"/qa1_dev100_designbyexample/QA1/UPDATES
  USING (SELECT case WHEN "mixedCaseCol" in (1,2) THEN 1 ELSE "mixedCaseCol" end "mixedCaseCol"FROM
/users/composite/test/sources/"db2_9.5"/qa1_dev100_designbyexample/mixedCaseSchema/mixedCaseTable)
mixedCaseTable
  ON c_id = mixedCaseCol
  WHEN MATCHED AND c_varchar = guid THEN DELETE
  WHEN NOT MATCHED THEN INSERT (c_id, c_varchar, c_decimal) VALUES (3, guid, 50);
  OPEN x FOR SELECT c_id, c_decimal FROM
/users/composite/test/sources/"db2_9.5"/qa1_dev100_designbyexample/QA1/UPDATES WHERE guid = c_varchar;
END

```

Example

This test is a NULL scan. Nothing should be executed.

```

PROC
PROCEDURE m_nullscan()
BEGIN
  MERGE INTO /users/composite/test/sources/oracle/DEV1/UPDATES
  USING /shared/examples/ds_inventory/tutorial/inventorytransactions
  ON 1<>1
  WHEN MATCHED THEN DELETE
  ;
END

```

Example

In this test, the left side of the JOIN is a physical selection.

```

PROC
PROCEDURE m_mixed_physical_selection()
BEGIN
  MERGE
  INTO /users/composite/test/sources/oracle/DEV1/UPDATES
  USING /shared/examples/ds_inventory/tutorial/inventorytransactions
  ON col_id = purchaseorderid AND col_char = pri_mp(781598358)
  WHEN MATCHED THEN UPDATE SET col_tinyint=productid;
  MERGE {option disable_push}
  INTO /users/composite/test/sources/oracle/DEV1/UPDATES
  USING /shared/examples/ds_inventory/tutorial/inventorytransactions

```

```

ON col_id = purchaseorderid AND col_char = pri_mp(781598358)
WHEN MATCHED THEN UPDATE SET col_tinyint=productid;
END

```

Example

This test verifies that MySQL requires the target table to have a unique index for all columns to be selected in that index.

```

PROC
PROCEDURE m_mixed()
BEGIN
MERGE INTO /users/composite/test/sources/mysql_v5/inventory/products
USING /users/composite/test/sources/mysql_v5/inventory/inventorytransactions
ON productname = transactiondescription
WHEN MATCHED THEN UPDATE SET categoryid = categoryid
;
END

```

Example

If the following SQL had used a SELECT statement, the logical plan generator would probably prune the left side. Using a MERGE prevents this from happening.

```

PROC
PROCEDURE m_outer_join_pruner()
BEGIN
MERGE
/users/composite/test/sources/mysql_v5/covoter/district USING
/users/composite/test/sources/mysql_v5/mysql/m10
ON
m10.m_id = district.oid
WHEN MATCHED THEN DELETE;
END

```

Example

The following MERGE is actually a no-op scan. No rows are matched, and there is no WHEN NOT MATCHED clause. The query engine should replace it with a no-op scan operator.

```

PROC
PROCEDURE null_scan()
BEGIN
MERGE INTO /users/composite/test/sources/oracle/DEV1/UPDATES u
USING /shared/examples/ds_inventory/tutorial/products p
ON 1 = 2
WHEN MATCHED THEN DELETE
;
END

```

ORDER BY Clause

An ORDER BY clause can return results in a different order when pushed vs. not pushed. For example, TDV returns NULLs first and considers the unary minus-sign when ordering floating-point numbers.

SPACE Function

Depending on where it is executed, the SPACE function with negative arguments can return different results. For example, for SPACE(-1):

- TDV (function not pushed) returns NULL.
- Microsoft SQL Server returns NULL.
- DB2 throws an exception.
- Greenplum, MySQL, PostgreSQL, and Vertica return nothing.

SQL Server Sorting Order

SQL Server supports multiple collating schemes, and its default is not the same as TDV. Furthermore, TDV cannot change data source collating schemes connection by connection.

The default SQL Server collating behavior results in incorrect results when columns contain special characters in situations like this:

- SQL Server data is on one side of a SORT MERGE join algorithm. The query engine inserts an ORDER BY clause on the joining columns, and the orderings differ.

An option is to use {OPTION HASH} in SORT MERGE queries, forcing TDV to use a HASH algorithm instead of SORT MERGE for joins. Be aware, though, that the HASH algorithm uses more memory because the query engine needs to hash the smaller side and then streams the bigger side over it.

- SQL Server data is in a comparison predicate of a WHERE clause.
- SQL Server data is in an ORDER BY clause.

In many situations you can specify a different collating scheme in the SQL (for example, using "COLLATE Latin1_General_BIN"), but this can interfere with indexing and thus affect performance.

Time Functions

When TDV deals with data types such as TIME or TIMESTAMP that are combined with TIMEZONE, TDV applies the TIMEZONE offset to the TIME or TIMESTAMP, but the original time zone information is then lost as the data is further manipulated.

The fractional-second precision of a returned TIMESTAMP value (milliseconds, microseconds, and so on) might differ depending on whether a query is pushed or not, or which data source processes the query.

Truncation vs. Rounding

TDV truncates values to the right of the decimal point when converting a NUMERIC, DECIMAL, FLOAT, or DOUBLE to an INTEGER type. Some data sources do rounding; others match TDV behavior. The SQL standard leaves implementation up to the vendor.

Because of this difference, results can differ when:

- Functions are applied that perform such conversions
- Numeric data is CAST to an INTEGER type
- Type promotion is performed during caching

In most cases, the TDV query engine warns the user when it detects a mismatch of this kind. However, the query engine cannot detect all such mismatches, and the query engine cannot normalize data source behavior for federated queries.

TDV Native Function Support

TDV *as a data source* supports the following types of functions:

- [TDV Aggregate Function Support, page 532](#)
- [TDV Character Function Support, page 532](#)
- [TDV Conditional Function Support, page 533](#)
- [TDV Conversion Function Support, page 533](#)
- [TDV Date Function Support, page 533](#)
- [TDV Numeric Function Support, page 534](#)

TDV Aggregate Function Support

TDV *as a data source* supports the aggregate functions listed in the table below.

| TDV Aggregate Function | Notes |
|------------------------|-------|
| AVG | |
| COUNT | |
| LISTAGG | |
| MAX | |
| MIN | |
| PERCENTILE_CONT | |
| PERCENTILE_DISC | |
| SUM | |
| VARIANCE_POP | |
| VARIANCE_SAMP | |

TDV Character Function Support

TDV *as a data source* supports the character functions listed in the table below.

| TDV Character Function | Notes |
|------------------------|-------|
| CONCAT | |
| LENGTH | |
| LOWER | |
| POSITION | |
| REPLACE | |
| RTRIM | |
| SUBSTRING | |
| TRIM | |

| TDV Character Function | Notes |
|------------------------|-------|
| UPPER | |

TDV Conditional Function Support

TDV *as a data source* supports the conditional function listed in the table below.

| TDV Conditional Function | Notes |
|--------------------------|-------|
| NULLIF | |

TDV Conversion Function Support

TDV *as a data source* supports the conversion functions listed in the table below.

| TDV Conversion Function | Notes |
|-------------------------|-------|
| CAST | |
| TO_CHAR | |
| TO_DATE | |
| TO_NUMBER | |
| TO_TIMESTAMP | |

TDV Date Function Support

TDV *as a data source* supports the date functions listed in the table below.

| TDV Date Function | Notes |
|-------------------|-------|
| YEAR | |

TDV Numeric Function Support

TDV *as a data source* supports the numeric functions listed in the table below.

| TDV Numeric Function | Notes |
|----------------------|-------|
| ABS | |
| ACOS | |
| ASIN | |
| ATAN | |
| CEILING | |
| COS | |
| COT | |
| DEGREES | |
| EXP | |
| FLOOR | |
| LOG | |
| PI | |
| POWER | |
| RADIANS | |
| ROUND | |
| SIN | |
| SQRT | |
| TAN | |

DataDirect Mainframe Function Support

TDV supports the following types of functions for DataDirect Mainframe:

- [DataDirect Mainframe Aggregate Function Support, page 535](#)
- [DataDirect Mainframe Character Function Support, page 535](#)
- [DataDirect Mainframe Conditional Function Support, page 536](#)
- [DataDirect Mainframe Conversion Function Support, page 536](#)
- [DataDirect Mainframe Date Function Support, page 537](#)
- [DataDirect Mainframe Numeric Function Support, page 537](#)
- [DataDirect Mainframe XML Function Support, page 538](#)

DataDirect Mainframe Aggregate Function Support

TDV supports the aggregate functions listed in the table below for DataDirect Mainframe.

| DataDirect Mainframe Aggregate Function | Notes |
|---|---|
| AVG | BLOB, CLOB, and string-type arguments not supported. |
| COUNT | BLOB and CLOB arguments not supported. |
| MAX | LONGVARCHAR argument can cause an exception; BLOB and CLOB arguments not supported. |
| MIN | LONGVARCHAR argument can cause an exception; BLOB and CLOB arguments not supported. |
| SUM | BLOB and CLOB arguments not supported. |

DataDirect Mainframe Character Function Support

TDV supports the character functions listed in the table below for DataDirect Mainframe.

| DataDirect Mainframe Character Function | Notes |
|---|--------------------------------------|
| CONCAT | LONGVARCHAR arguments not supported. |
| LENGTH | |

| DataDirect Mainframe Character Function | Notes |
|---|-------|
| LOWER | |
| POSITION | |
| REPLACE | |
| RTRIM | |
| SPACE | |
| SUBSTRING | |
| TRIM | |
| UPPER | |

DataDirect Mainframe Conditional Function Support

TDV supports the conditional function listed in the table below for DataDirect Mainframe.

| DataDirect Mainframe Conditional Function | Notes |
|---|---|
| NULLIF | NULL not supported; BLOB, CLOB, LONGVARCHAR_FOR_BIT_DATA, LONG_VARCHAR arguments not allowed. |

DataDirect Mainframe Conversion Function Support

TDV supports the conversion functions listed in the table below for DataDirect Mainframe.

| DataDirect Mainframe Conversion Function | Notes |
|--|-------|
| CAST | |
| FORMAT | |
| PARSE_TIMESTAMP | |

| DataDirect Mainframe Conversion Function | Notes |
|--|-------|
| TO_CHAR | |
| TO_DATE | |
| TO_NUMBER | |
| TO_TIMESTAMP | |

DataDirect Mainframe Date Function Support

TDV supports the date functions listed in the table below for DataDirect Mainframe.

| DataDirect Mainframe Date Function | Notes |
|------------------------------------|--|
| CURRENT_DATE | |
| CURRENT_TIME | |
| CURRENT_TIMESTAMP | |
| DAY | |
| MONTH | |
| YEAR | Version >8, or DB2 XML Extender enabled. |

DataDirect Mainframe Numeric Function Support

TDV supports the numeric functions listed in the table below for DataDirect Mainframe.(PI is not supported.)

| DataDirect Mainframe Numeric Function | Notes |
|---------------------------------------|-------|
| ABS | |
| ACOS | |
| ASIN | |

| DataDirect Mainframe Numeric Function | Notes |
|---------------------------------------|-------|
| ATAN | |
| CEILING | |
| COS | |
| COT | |
| DEGREES | |
| EXP | |
| FLOOR | |
| LOG | |
| POWER | |
| RADIANS | |
| ROUND | |
| SIN | |
| SQRT | |
| TAN | |

DataDirect Mainframe XML Function Support

TDV supports the XML functions listed in the table below for DataDirect Mainframe.

The XML functions can be used only if all DB2 data sources of version 8 or earlier have DB2 XML Extender enabled.

| DataDirect Mainframe XML Function | Notes |
|-----------------------------------|-------|
| XMLATTRIBUTES | |
| XMLCOMMENT | |
| XMLCONCAT | |

| DataDirect Mainframe XML Function | Notes |
|-----------------------------------|-------|
| XMLDOCUMENT | |
| XMLELEMENT | |
| XMLFOREST | |
| XMLNAMESPACES | |
| XMLPI | |
| XMLQUERY | |
| XMLTEXT | |

DB2 Function Support

If the TDV and DB2 data source settings do not match for case sensitivity or trailing spaces, use the STRICT option in any query that includes a DISTINCT operator.

TDV supports the following types of functions for DB2:

- [DB2 Aggregate Function Support, page 540](#)
- [DB2 Analytic Function Support, page 540](#)
- [DB2 Analytic Aggregate Function Support, page 541](#)
- [DB2 Character Function Support, page 541](#)
- [DB2 Conditional Function Support, page 542](#)
- [DB2 Conversion Function Support, page 543](#)
- [DB2 Date Function Support, page 543](#)
- [DB2 Linear Regression Function Support, page 544](#)
- [DB2 Numeric Function Support, page 545](#)
- [DB2 XML Function Support, page 546](#)

DB2 Aggregate Function Support

TDV supports the aggregate functions listed in the table below for DB2.

| DB2 Aggregate Function | Notes |
|------------------------|---|
| AVG | BLOB, CLOB, and string-type arguments not supported. |
| CORR | BLOB and CLOB arguments not supported; arguments must be numeric. |
| COUNT | BLOB and CLOB arguments not supported; DISTINCT not supported with LONGVAR. |
| MAX | BLOB and CLOB arguments not supported. |
| MIN | BLOB and CLOB arguments not supported. |
| SUM | BLOB and CLOB arguments not supported. |

DB2 Analytic Function Support

TDV supports the analytic functions listed in the table below for DB2..

| DB2 Analytic Function | Notes |
|-----------------------|--|
| AVG | AVG DISTINCT not supported for versions 8 and 9. |
| COUNT | COUNT DISTINCT not supported for versions 8 and 9. |
| CUME_DIST | Supported only for version 11. |
| DENSE_RANK | |
| MAX | MAX DISTINCT not supported for versions 8 and 9. |
| MIN | MIN DISTINCT not supported for versions 8 and 9. |
| PERCENT_RANK | |
| RANK | |

| DB2 Analytic Function | Notes |
|-----------------------|--|
| ROW_NUMBER | |
| SUM | SUM DISTINCT not supported for versions 8 and 9. |

DB2 Analytic Aggregate Function Support

TDV supports the analytic aggregate functions listed in the table below for DB2.

| DB2 Analytic Aggregate Function | Notes |
|---------------------------------|---|
| MEDIAN | Supported only for version 11. |
| PERCENTILE_CONT | Supported only for version 11. |
| PERCENTILE_DISC | Supported only for version 11. |
| STDDEV | Supported only for version 11. Note: To prevent nonstandard results, add OPTION STRICT to the query. |
| STDDEV_POP | |
| STDDEV_SAMP | Not supported for version 8. |
| VARIANCE_POP | |
| VARIANCE_SAMP | Not supported for version 8. |

DB2 Character Function Support

TDV supports the character functions listed in the table below for DB2.

| DB2 Character Function | Notes |
|------------------------|--------------------------------|
| BTRIM | Supported only for version 11. |
| CONCAT | LONGVARCHAR not supported. |
| LEFT | Supported only for version 11. |

| DB2 Character Function | Notes |
|------------------------|--------------------------------------|
| LENGTH | |
| LOWER | |
| POSITION | |
| REGEXP_REPLACE | Supported only for version 11. |
| REPLACE | |
| RIGHT | Supported only for version 11. |
| RTRIM | |
| SPACE | SMALLINT and INTEGER arguments only. |
| STRPOS | Supported only for version 11. |
| SUBSTRING | |
| TRIM | |
| UPPER | |

DB2 Conditional Function Support

TDV supports the conditional functions listed in the table below for DB2.

| DB2 Conditional Function | Notes |
|--------------------------|--|
| COALESCE | |
| DECODE | Mapped to CASE. |
| NULLIF | NULL not supported; BLOB, CLOB, LONGVARCHAR_FOR_BIT_DATA, LONG_VARCHAR arguments not allowed. For string comparisons, ignores trailing spaces. |

DB2 Conversion Function Support

TDV supports the conversion functions listed in the table below for DB2.

| DB2 Conversion Function | Notes |
|-------------------------|---|
| CAST | The maximum length for BINARY and VARBINARY is 4000. The maximum length for CHAR is 254. The maximum length of precision (p) and scale (s) is 31. |
| FORMAT_DATE | |
| PARSE_TIMESTAMP | |
| TO_CHAR | |
| TO_DATE | |
| TO_NUMBER | Empty-string input returns an exception. |
| TO_TIMESTAMP | Input of a string of eight white spaces returns a timestamp value. Shorter input string or a trimmed value throws an exception. |

DB2 Date Function Support

TDV supports the date functions listed in the table below for DB2.

| Function | Notes |
|-------------------|---|
| ADD_DAYS | Supported only for version 11. |
| ADD_MONTHS | Result differs between pushed and not pushed. When pushed, if expression is the last day of month or if the resulting month has fewer days than the day component of the expression, the result is the last day of the resulting month. When not pushed, the result has the same day component as the expression. |
| CURRENT_DATE | |
| CURRENT_TIME | |
| CURRENT_TIMESTAMP | |

| Function | Notes |
|--------------|--|
| DATE_ADD | Supported only for version 11. |
| DATE_PART | Supported only for version 11. |
| DATE_TRUNC | Supported only for version 11. |
| DAY | |
| DAYOFMONTH | Supported only for version 11. |
| DAYS_BETWEEN | Supported only for version 11. |
| MONTH | |
| NOW | Supported only for version 11. |
| YEAR | Version >8, or DB2 XML Extender enabled. |

DB2 Linear Regression Function Support

TDV supports the linear regression functions listed in the table below for DB2 version 11..

| DB2 Linear Regression Function | Notes |
|--------------------------------|-------|
| REGR_SLOPE | |
| REGR_INTERCEPT | |
| REGR_COUNT | |
| REGR_R2 | |
| REGR_AVGX | |
| REGR_SXX | |
| REGR_SYY | |
| REGR_SXY | |

DB2 Numeric Function Support

TDV supports the numeric functions listed in the table below for DB2. DB2 does not support string-type arguments in numeric functions.

| DB2 Numeric Function | Notes |
|----------------------|--------------------------------|
| ABS | |
| ACOS | |
| ASIN | |
| ATAN | |
| CEILING | |
| COS | |
| COT | |
| DEGREES | |
| EXP | |
| FLOOR | |
| LOG | |
| PI | Not supported. |
| POW | Supported only for version 11. |
| POWER | |
| RADIANS | |
| RANDOM | |
| ROUND | |
| SIN | |
| SQRT | |
| TAN | |

DB2 XML Function Support

TDV supports the XML functions listed in the table below for DB2.

The XML functions can be used only if all DB2 data sources of version 8 or earlier have DB2 XML Extender enabled; otherwise an exception will be thrown.

| DB2 XML Function | Notes |
|------------------|-------|
| XMLATTRIBUTES | |
| XMLCOMMENT | |
| XMLCONCAT | |
| XMLDOCUMENT | |
| XMLELEMENT | |
| XMLFOREST | |
| XMLNAMESPACES | |
| XMLPI | |
| XMLQUERY | |
| XMLTEXT | |

DB2 Mainframe Function Support

TDV supports the functions listed in the table below for DB2 Mainframe:

- [DB2 Mainframe Aggregate Function Support, page 547](#)
- [DB2 Mainframe Character Function Support, page 548](#)
- [DB2 Mainframe Conditional Function Support, page 548](#)
- [DB2 Mainframe Conversion Function Support, page 549](#)
- [DB2 Mainframe Date Function Support, page 549](#)
- [DB2 Mainframe Numeric Function Support, page 551](#)
- [DB2 Mainframe XML Function Support, page 552](#)

DB2 Mainframe Aggregate Function Support

TDV supports the aggregate functions listed in the table below for DB2 Mainframe.

| DB2 Mainframe Aggregate Function | Notes |
|----------------------------------|--|
| AVG | BLOB, CLOB, and string-type arguments not supported. |
| COUNT | BLOB and CLOB arguments not supported. |
| MAX | BLOB and CLOB arguments not supported. |
| MIN | BLOB and CLOB arguments not supported. |
| SUM | BLOB and CLOB arguments not supported. |

DB2 Mainframe Character Function Support

TDV supports the character functions listed in the table below for DB2 Mainframe.

| DB2 Mainframe Character Function | Notes |
|----------------------------------|--------------------------------------|
| CONCAT | LONGVARCHAR not supported. |
| LENGTH | |
| LOWER | |
| POSITION | String-type arguments only. |
| REPLACE | |
| RTRIM | |
| SPACE | SMALLINT and INTEGER arguments only. |
| SUBSTRING | |
| TRIM | |
| UPPER | |

| DB2 Mainframe Character Function | Notes |
|----------------------------------|--|
| SUBSTR | |
| ASCII | |
| INSERT | |
| LPAD | Result differs between pushed and not pushed. When pushed, the length argument will not accept non-constant value. |
| RPAD | Result differs between pushed and not pushed. When pushed, the length argument will not accept non-constant value. |
| DIFFERENCE | |
| TRIM | |
| REPEAT | |

DB2 Mainframe Conditional Function Support

TDV supports the conditional function listed in the table below for DB2 Mainframe.

| DB2 Mainframe Conditional Function | Notes |
|------------------------------------|--|
| NULLIF | NULL not supported; BLOB, CLOB, LONGVARCHAR_FOR_BIT_DATA, LONG_VARCHAR arguments not allowed. For string comparisons, ignores trailing spaces. |
| IFNULL | |
| DECODE | |
| NVL | |

DB2 Mainframe Binary Function Support

TDV supports the binary functions listed in the table below for DB2 Mainframe

| DB2 Mainframe Conditional Function | Notes |
|--|-------|
| BITAND, BITANDNOT, BITOR, BITXOR, BITNOT | |

DB2 Mainframe Conversion Function Support

TDV supports the conversion functions listed in the table below for DB2 Mainframe.

| DB2 Mainframe Conversion Function | Notes |
|-----------------------------------|---|
| CAST | The maximum length for BINARY and VARBINARY is 4000. The maximum length for CHAR is 254. The maximum length of precision (p) and scale (s) is 31. |
| TO_CHAR | |
| TO_DATE | |
| TO_NUMBER | |
| TO_TIMESTAMP | |
| TO_TIMESTAMP_TZ | |

DB2 Mainframe Date Function Support

TDV supports the date functions listed in the table below for DB2 Mainframe.

| DB2 Mainframe Date Function | Notes |
|-----------------------------|-------|
| CURRENT_DATE | |
| CURRENT_TIME | |
| CURRENT_TIMESTAMP | |

| DB2 Mainframe Date Function | Notes |
|-----------------------------|-------|
| DAY | |
| MONTH | |
| YEAR | |
| ADD_MONTHS | |
| DAYOFMONTH | |
| DAYOFWEEK | |
| DAYOFWEEK_ISO | |
| DAYOFYEAR | |
| DAYS | |
| HOUR | |
| JULIAN_DAY | |
| LAST_DAY | |
| MICROSECOND | |
| MIDNIGHT_SECONDS | |
| MINUTE | |
| MONTHS_BETWEEN | |
| NEXT_DAY | |
| QUARTER | |
| SECOND | |
| TIMESTAMP | |
| WEEK | |
| WEEK_ISO | |
| EXTRACT | |

DB2 Mainframe Numeric Function Support

TDV supports the numeric functions listed in the table below for DB2 Mainframe. DB2 Mainframe does not support string-type arguments in numeric functions.

| DB2 Mainframe Numeric Function | Notes |
|--------------------------------|----------------|
| ABS | |
| ACOS | |
| ASIN | |
| ATAN | |
| CEILING | |
| COS | |
| COT | |
| DEGREES | |
| EXP | |
| FLOOR | |
| LOG | |
| PI | Not supported. |
| POWER | |
| RADIANS | |
| ROUND | |
| SIN | |
| SQRT | |
| TAN | |
| UNICODE | |
| SIGN | |

| DB2 Mainframe Numeric Function | Notes |
|--------------------------------|-------|
| QUANTIZE | |
| DECFLOAT | |
| NORMALIZE_DECFLOAT | |
| ATAN2 | |
| LN | |
| MOD | |
| RAND | |

DB2 Mainframe XML Function Support

TDV supports the XML functions listed in the table below for DB2 Mainframe.

The XML functions can be used only if all DB2 data sources of version 8 or earlier have DB2 XML Extender enabled; otherwise an exception will be thrown.

| DB2 Mainframe XML Function | Notes |
|----------------------------|-------|
| XMLATTRIBUTES | |
| XMLCOMMENT | |
| XMLCONCAT | |
| XMLDOCUMENT | |
| XMLELEMENT | |
| XMLFOREST | |
| XMLNAMESPACES | |
| XMLPI | |
| XMLQUERY | |
| XMLTEXT | |

| DB2 Mainframe XML Function | Notes |
|----------------------------|-------|
| XMLAGG | |

File Function Support

TDV supports the following types of functions for file data sources:

- [File Aggregate Function Support, page 553](#)
- [File Character Function Support, page 553](#)
- [File Conversion Function Support, page 554](#)
- [File Date Function Support, page 554](#)
- [File Numeric Function Support, page 555](#)

File Aggregate Function Support

TDV supports the aggregate functions listed in the table below for file data sources.

| File Aggregate Function | Notes |
|-------------------------|-------|
| AVG | |
| COUNT | |
| MAX | |
| MIN | |
| SUM | |

File Character Function Support

TDV supports the character functions listed in the table below for file data sources.

| File Character Function | Notes |
|-------------------------|-------|
| CONCAT | |

| File Character Function | Notes |
|-------------------------|-------|
| LENGTH | |
| LOWER | |
| REPLACE | |
| RTRIM | |
| SUBSTRING | |
| TRIM | |
| UPPER | |

File Conversion Function Support

TDV supports the conversion functions listed in the table below for file data sources.

| File Conversion Function | Notes |
|--------------------------|-------|
| CAST | |
| TO_CHAR | |
| TO_DATE | |
| TO_NUMBER | |
| TO_TIMESTAMP | |

File Date Function Support

TDV supports the date functions listed in the table below for file data sources.

| File Date Function | Notes |
|--------------------|-------|
| CURDAY | |
| CURTIME | |
| CURTIMESTAMP | |

| File Date Function | Notes |
|--------------------|-------|
| DAY | |
| MONTH | |
| YEAR | |

File Numeric Function Support

TDV supports the numeric functions listed in the table below for file data sources.

| File Numeric Function | Notes |
|-----------------------|-------|
| ABS | |
| ACOS | |
| ASIN | |
| ATAN | |
| CEILING | |
| COS | |
| COT | |
| DEGREES | |
| EXP | |
| FLOOR | |
| LOG | |
| PI | |
| POWER | |
| RADIANS | |
| ROUND | |
| SIN | |
| SQRT | |

| File Numeric Function | Notes |
|-----------------------|-------|
| TAN | |

Greenplum Function Support

Greenplum Database is based on PostgreSQL and adheres to the same SQL structure and syntax (with minor exceptions).

TDV supports the following types of functions for Greenplum:

- [Greenplum Aggregate Function Support, page 556](#)
- [Greenplum Analytic Function Support, page 557](#)
- [Greenplum Analytic Aggregate Function Support, page 558](#)
- [Greenplum Binary Function Support, page 560](#)
- [Greenplum Character Function Support, page 561](#)
- [Greenplum Conditional Function Support, page 563](#)
- [Greenplum Conversion Function Support, page 564](#)
- [Greenplum Date Function Support, page 564](#)
- [Greenplum Numeric Function Support, page 566](#)
- [Greenplum Time Function Support, page 568](#)

Greenplum Aggregate Function Support

TDV supports the aggregate functions listed in the table below for Greenplum. DISTINCT is supported for all of these functions.

| Greenplum Aggregate Function | Notes |
|------------------------------|-----------------|
| AVG | Push supported. |
| BIT_AND | Push supported. |
| BIT_OR | Push supported. |
| COUNT | Push supported. |
| MAX | Push supported. |
| MIN | Push supported. |

| Greenplum Aggregate Function | Notes |
|------------------------------|-----------------|
| SUM | Push supported. |

Greenplum Analytic Function Support

TDV supports the analytic functions listed in the table below for Greenplum.

The following functions can not be pushed:

- EXP_WEIGHTED_AVG
- FIRST_VALUE_IGNORE_NULLS
- LAST_VALUE_IGNORE_NULLS
- NTH_VALUE_FROM_LAST
- NTH_VALUE_FROM_LAST_IGNORE_NULLS
- NTH_VALUE
- NTH_VALUE_IGNORE_NULLS
- RATIO_TO_REPORT
- TIMESERIES

| Greenplum Analytic Function | Notes |
|-----------------------------|-----------------|
| AVG | Push supported. |
| CORR | Push supported. |
| COUNT | Push supported. |
| COVAR_POP | Push supported. |
| COVAR_SAMP | Push supported. |
| CUME_DIST | Push supported. |
| DENSE_RANK | Push supported. |
| FIRST_VALUE | Push supported. |
| LAG | Push supported. |
| LAST_VALUE | Push supported. |

| Greenplum Analytic Function | Notes |
|-----------------------------|---|
| LEAD | Push supported. |
| MAX | Push supported. |
| MIN | Push supported. |
| NTILE | Push supported. |
| PERCENT_RANK | Push supported. |
| RANK | Push supported. |
| ROW_NUMBER | |
| STDDEV | DISTINCT supported. Push supported. TDV's implementation of STDDEV upcasts 32 bit float to 64 bit double. The result is a double |
| STDDEV_POP | Push supported. |
| STDDEV_SAMP | Push supported. |
| VAR_POP | Push supported. |
| VAR_SAMP | Push supported. |
| VARIANCE | DISTINCT supported. Push supported. |
| VARIANCE_POP | Push supported. |
| VARIANCE_SAMP | Push supported. |

Greenplum Analytic Aggregate Function Support

TDV supports the analytic aggregate functions listed in the table below for Greenplum.

The following functions can not be pushed:

- CORR_SPEARMAN
- LISTAGG

- MEDIAN
- PERCENTILE_CONT
- PERCENTILE_DISC
- XMLAGG

| Greenplum Analytic Aggregate Function | Notes |
|---------------------------------------|-----------------|
| CORR | Push supported. |
| COVAR_POP | Push supported. |
| COVAR_SAMP | Push supported. |
| REGR_AVGX | Push supported. |
| REGR_AVGY | Push supported. |
| REGR_COUNT | Push supported. |
| REGR_INTERCEPT | Push supported. |
| REGR_R2 | Push supported. |
| REGR_SLOPE | Push supported. |
| REGR_SXX | Push supported. |
| REGR_SXY | Push supported. |
| REGR_SYY | Push supported. |
| STDDEV | Push supported. |
| STDDEV_POP | Push supported. |
| STDDEV_SAMP | Push supported. |
| VAR_POP | Push supported. |
| VAR_SAMP | Push supported. |
| VARIANCE | Push supported. |
| VARIANCE_POP | Push supported. |

| Greenplum Analytic Aggregate Function | Notes |
|---------------------------------------|-----------------|
| VARIANCE_SAMP | Push supported. |

Greenplum Binary Function Support

TDV supports the binary functions listed in the table below for Greenplum.

| Greenplum Binary Function | Notes |
|---------------------------|-----------------|
| INT1AND | Push supported. |
| INT2AND | Push supported. |
| INT4AND | Push supported. |
| INT8AND | Push supported. |
| INT1OR | Push supported. |
| INT2OR | Push supported. |
| INT4OR | Push supported. |
| INT8OR | Push supported. |
| INT1SHL | Push supported. |
| INT2SHL | Push supported. |
| INT4SHL | Push supported. |
| INT8SHL | Push supported. |
| INT1SHR | Push supported. |
| INT2SHR | Push supported. |
| INT4SHR | Push supported. |
| INT8SHR | Push supported. |
| INT1XOR | Push supported. |
| INT2XOR | Push supported. |

| Greenplum Binary Function | Notes |
|---------------------------|-----------------|
| INT4XOR | Push supported. |
| INT8XOR | Push supported. |
| INT1NOT | Push supported. |
| INT2NOT | Push supported. |
| INT4NOT | Push supported. |
| INT8NOT | Push supported. |

Greenplum Character Function Support

TDV supports the character functions listed in the table below for Greenplum

The following functions can not be pushed:

- DLE_DST
- INSERT
- LE_DST
- LOCATE
- PARTIAL_STRING_MASK

| Greenplum Character Function | Notes |
|------------------------------|-----------------|
| ASCII | Push supported. |
| BIT_LENGTH | Push supported. |
| BTRIM | Push supported. |
| CHAR_LENGTH | Push supported. |
| CHARACTER_LENGTH | Push supported. |
| CHR | Push supported. |

| Greenplum Character Function | Notes |
|------------------------------|---|
| CONCAT | Results might differ between pushed and not pushed, even if the Ignore Trailing Space setting of the Greenplum data source is the same as that of TDV, because the Greenplum database always trims trailing spaces. |
| FIND | Push supported. |
| INITCAP | Push supported. |
| INSTR | Push supported. |
| LCASE | Push supported. |
| LENGTH | Push supported. |
| LOWER | Push supported. |
| LPAD | Push supported. |
| LTRIM | Push supported. |
| POSITION | Results might differ between pushed and not pushed, even if the Ignore Trailing Space setting of the Greenplum data source is the same as that of TDV, because the Greenplum database always trims trailing spaces. |
| REPEAT | Push supported. |
| REPLACE | Results might differ between pushed and not pushed, even if the Ignore Trailing Space setting of the Greenplum data source is the same as that of TDV, because the Greenplum database always trims trailing spaces. |
| RPAD | Push supported. |
| RTRIM | Push supported. |
| SPACE | Push supported. |
| STRPOS | Push supported. |
| SUBSTR | Push supported. |

| Greenplum Character Function | Notes |
|------------------------------|-----------------|
| SUBSTRING | Push supported. |
| TO_HEX | Push supported. |
| TRANSLATE | Push supported. |
| TRIM | Push supported. |
| TRIM(LEADING FROM) | Push supported. |
| TRIM(TRAILING FROM) | Push supported. |
| UCASE | Push supported. |
| UNICHR | Push supported. |
| UNICODE | Push supported. |
| UPPER | Push supported. |

Greenplum Conditional Function Support

TDV supports the conditional functions listed in the table below for Greenplum.

The following functions can not be pushed:

- DECODE
- ISNULL
- ISNUMERIC

| Greenplum Conditional Function | Notes |
|--------------------------------|-----------------|
| COALESCE | Push supported. |
| GREATEST | Push supported. |
| IFNULL | Push supported. |
| LEAST | Push supported. |
| NULLIF | Push supported. |
| NVL | Push supported. |

| Greenplum Conditional Function | Notes |
|--------------------------------|-----------------|
| NVL2 | Push supported. |

Greenplum Conversion Function Support

TDV supports the conversion functions listed in the table below for Greenplum. The following functions can not be pushed:

- PARSE_DATE
- PARSE_TIME
- TIMESTAMP
- TO_TIMESTAMP_TZ

| Greenplum Conversion Function | Notes |
|-------------------------------|-----------------|
| CAST | Push supported. |
| PARSE_TIMESTAMP | Push supported. |
| TO_CHAR | Push supported. |
| TO_DATE | Push supported. |
| TO_NUMBER | Push supported. |
| TO_TIMESTAMP | Push supported. |

Greenplum Date Function Support

TDV supports the date functions listed in the table below for Greenplum. The following functions can not be pushed:

- DATEADD
- DATENAME
- DAYNAME
- DAYS_BETWEEN
- DBTIMEZONE
- LAST_DAY
- MONTHS_BETWEEN

- NEW_TIME
- NEXT_DAY
- NUMTODSINTERVAL
- NUMTOYMINTERVAL
- TIME_SLICE
- TRUNC
- TZCONVERTOR
- UTC_TO_TIMESTAMP
- .

| Greenplum Date Function | Notes |
|-------------------------|-----------------|
| ADD_MONTHS | Push supported. |
| CLOCK_TIMESTAMP | Push supported. |
| CURRENT_DATE | Push supported. |
| CURRENT_TIME | Push supported. |
| CURRENT_TIMESTAMP | Push supported. |
| DATE_ADD | Push supported. |
| DATE_PART | Push supported. |
| DATE_SUB | Push supported. |
| DATE_TRUNC | Push supported. |
| DATEPART | Push supported. |
| DATETRUNC | Push supported. |
| DAY | Push supported. |
| EXTRACT(DAY FROM) | Push supported. |
| EXTRACT(DOW FROM) | Push supported. |
| EXTRACT(DOY FROM) | Push supported. |
| EXTRACT(EPOCH FROM) | Push supported. |

| Greenplum Date Function | Notes |
|----------------------------|-----------------|
| EXTRACT(HOUR FROM) | Push supported. |
| EXTRACT(MICROSECOND FROM) | Push supported. |
| EXTRACT(MILLISECOND FROM) | Push supported. |
| EXTRACT(MINUTE FROM) | Push supported. |
| EXTRACT(MONTH FROM) | Push supported. |
| EXTRACT(QUARTER FROM) | Push supported. |
| EXTRACT(SECOND FROM) | Push supported. |
| EXTRACT(WEEK FROM) | Push supported. |
| EXTRACT(YEAR FROM) | Push supported. |
| FORMAT_DATE | Push supported. |
| LOCALTIME | Push supported. |
| LOCALTIMESTAMP | Push supported. |
| MONTH | Push supported. |
| NOW | Push supported. |
| TIMEOFDAY | Push supported. |
| YEAR | Push supported. |

Greenplum Numeric Function Support

TDV supports the numeric functions listed in the table below for Greenplum. The following functions can not be pushed:

- COSH
- SINH
- TANH
- FACTORIAL
- ROWNUM

| Greenplum Numeric Function | Notes |
|----------------------------|-----------------|
| ABS | Push supported. |
| ACOS | Push supported. |
| ASIN | Push supported. |
| ATAN | Push supported. |
| ATAN2 | Push supported. |
| CBRT | Push supported. |
| CEIL | Push supported. |
| CEILING | Push supported. |
| COS | Push supported. |
| COT | Push supported. |
| DEGREES | Push supported. |
| EXP | Push supported. |
| FLOOR | Push supported. |
| LN | Push supported. |
| LOG | Push supported. |
| MOD | Push supported. |
| NUMERIC_LOG | Push supported. |
| PI() | Push supported. |
| POW | Push supported. |
| POWER | Push supported. |
| RADIANS | Push supported. |
| RAND | Push supported. |

| Greenplum Numeric Function | Notes |
|----------------------------|-----------------|
| RANDOM | Push supported. |
| ROUND | Push supported. |
| SIGN | Push supported. |
| SIN | Push supported. |
| SQRT | Push supported. |
| TAN | Push supported. |
| TRUNC | Push supported. |

Greenplum Time Function Support

TDV supports the time function listed in the table below for Greenplum.

| Greenplum Time Function | Notes |
|-------------------------|-------|
| EXTRACT | |

HBase Function Support

TDV supports the following types of functions for HBase:

- [HBase Aggregate Function Support, page 569](#)
- [HBase Conversion Function Support, page 569](#)
- [HBase Date Function Support, page 569](#)
- [HBase Numeric Function Support, page 570](#)
- [HBase String Function Support, page 570](#)

HBase Aggregate Function Support

TDV supports the aggregate functions listed in the table below for HBase.

| HBase Aggregate Function | Notes |
|--------------------------|-------|
| AVG | |
| COUNT | |
| MAX | |
| MIN | |
| STDDEV_POP | |
| STDDEV_SAMP | |
| SUM | |

HBase Conversion Function Support

TDV supports the conversion functions listed in the table below for HBase.

| HBase Conversion Function | Notes |
|---------------------------|-------|
| CAST | |
| TO_CHAR | |

HBase Date Function Support

TDV supports the date functions listed in the table below for HBase.

| HBase Date Function | Notes |
|---------------------|-------|
| COALESCE | |
| CURRENT_DATE | |
| CURRENT_TIME | |

HBase Numeric Function Support

TDV supports the numeric functions listed in the table below for HBase.

| HBase Numeric Function | Notes |
|------------------------|-------|
| CEIL | |
| CEILING | |
| FLOOR | |
| ROUND | |

HBase String Function Support

TDV supports the string functions listed in the table below for HBase.

| HBase String Function | Notes |
|-----------------------|-------|
| CONCAT | |
| LCASE | |
| LENGTH | |
| LOWER | |
| LTRIM | |
| RTRIM | |
| SUBSTR | |
| UCASE | |
| UPPER | |

HSQLDB Function Support

TDV supports the following types of functions for HSQLDB DB:

- [HSQLDB Aggregate Function Support, page 571](#)

- [HSQLDB Binary Function Support, page 572](#)
- [HSQLDB Conversion Function Support, page 572](#)
- [HSQLDB Date Function Support, page 572](#)
- [HSQLDB Numeric Function Support, page 573](#)
- [HSQLDB String Function Support, page 574](#)

HSQLDB Aggregate Function Support

TDV supports the aggregate functions listed in the table below for HSQLDB.

| HSQLDB Aggregate Function | Notes |
|---------------------------|-------|
| AVG | |
| CORR | |
| COUNT | |
| COVAR_POP | |
| COVAR_SAMP | |
| MAX | |
| MIN | |
| SPACE | |
| STDDEV | |
| STDDEV_POP | |
| STDDEV_SAMP | |
| SUM | |
| VAR_POP | |
| VAR_SAMP | |
| VARIANCE | |

HSQLDB Binary Function Support

TDV supports the binary functions listed in the table below for HSQLDB

| HSQLDB Binary Function | Notes |
|------------------------------------|-------|
| INT1AND, INT2AND, INT4AND, INT8AND | |
| INT1NOT, INT2NOT, INT4NOT, INT8NOT | |
| INT1OR, INT2OR, INT4OR, INT8OR | |
| INT1XOR, INT2XOR, INT4XOR, INT8XOR | |

HSQLDB Conversion Function Support

TDV supports the conversion functions listed in the table below for HSQLDB

| HSQLDB Conversion Function | Notes |
|----------------------------|------------------------------------|
| CAST | Maximum VARCHAR length: 2147483647 |
| TO_CHAR | |

HSQLDB Date Function Support

TDV supports the date functions listed in the table below for HSQLDB

| HSQLDB Date Function | Notes |
|----------------------|-------|
| COALESCE | |
| CURRENT_DATE | |
| CURRENT_TIME | |
| CURRENT_TIMESTAMP | |
| DATE_ADD | |
| DATE_SUB | |

| HSQLDB Date Function | Notes |
|----------------------|-------|
| DATEDIFF | |
| DAY | |
| EXTRACT | |
| FROM_UNIXTIME | |
| MONTH | |
| NOW | |
| TIMEOFDAY | |
| TO_DATE | |
| TO_TIMESTAMP | |
| UNIX_TIMESTAMP | |
| UTC_TO_TIMESTAMP | |
| YEAR | |

HSQLDB Numeric Function Support

TDV supports the numeric functions listed in the table below for HSQLDB

| HSQLDB Numeric Function | Notes |
|-------------------------|-------|
| ABS | |
| ACOS | |
| ASIN | |
| ATAN | |
| CEIL | |
| CEILING | |
| COS | |
| DEGREES | |

| HSQLDB Numeric Function | Notes |
|-------------------------|-------|
| EXP | |
| FLOOR | |
| LN | |
| LOG | |
| PI | |
| POW | |
| POWER | |
| RADIANS | |
| RANDOM | |
| ROUND | |
| SIN | |
| SQRT | |
| TAN | |

HSQLDB String Function Support

TDV supports the string functions listed in the table below for HSQLDB

| HSQLDB String Function | Notes |
|------------------------|-------|
| ASCII | |
| CONCAT | |
| LCASE | |
| LENGTH | |
| LOWER | |
| LPAD | |
| LTRIM | |

| HSQLDB String Function | Notes |
|------------------------|--|
| REPLACE | |
| RPAD | |
| RTRIM | |
| SPACE | |
| SUBSTR | |
| SUBSTRING | Some data type combinations are not supported. |
| TRIM | |
| UCASE | |
| UPPER | |

Impala Function Support

TDV supports the following types of functions for Impala:

- [Impala Aggregate Function Support, page 576](#)
- [Impala Binary Function Support, page 576](#)
- [Impala Conditional Function Support, page 577](#)
- [Impala Conversion Function Support, page 577](#)
- [Impala Date Function Support, page 577](#)
- [Impala Numeric Function Support, page 578](#)
- [Impala Push-Only Function Support, page 580](#)
- [Impala String Function Support, page 580](#)

Impala Aggregate Function Support

TDV supports the aggregate functions listed in the table below for Impala.

| Impala Aggregate Function | Notes |
|---------------------------|-------|
| AVG | |
| CORR | |
| COUNT | |
| COVAR_POP | |
| COVAR_SAMP | |
| MAX | |
| MIN | |
| SPACE | |
| STDDEV | |
| STDDEV_POP | |
| STDDEV_SAMP | |
| SUM | |
| VAR_POP | |
| VAR_SAMP | |
| VARIANCE | |

Impala Binary Function Support

TDV supports the binary functions listed in the table below for Impala.

| Impala Binary Function | Notes |
|------------------------------------|-------|
| INT1AND, INT2AND, INT4AND, INT8AND | |
| INT1NOT, INT2NOT, INT4NOT, INT8NOT | |
| INT1OR, INT2OR, INT4OR, INT8OR | |

| Impala Binary Function | Notes |
|------------------------------------|-------|
| INT1XOR, INT2XOR, INT4XOR, INT8XOR | |

Impala Conditional Function Support

TDV supports the conditional function listed in the table below for Impala.

| Impala Conditional Function | Notes |
|-----------------------------|-------|
| COALESCE | |

Impala Conversion Function Support

TDV supports the conversion functions listed in the table below for Impala.

| Impala Conversion Function | Notes |
|----------------------------|--|
| CAST | Impala and TDV have different logic to handle cast(float). |
| TO_CHAR | Impala omits precision for to_char(double) function, if the last digital number is zero. |

Impala Date Function Support

TDV supports the date functions listed in the table below for Impala.

| Impala Date Function | Notes |
|----------------------|-------|
| CURRENT_DATE | |
| CURRENT_TIME | |
| CURRENT_TIMESTAMP | |
| DATE_ADD | |
| DATE_SUB | |
| DATEDIFF | |
| DAY | |

| Impala Date Function | Notes |
|----------------------|--|
| EXTRACT | TDV and Impala have different logic when extracting second from timestamp. TDV includes second+millisecond. Impala does not include millisecond information. |
| FROM_UNIXTIME | |
| MINUTE | |
| MONTH | |
| NOW | |
| SECOND | |
| TIMEOFDAY | |
| TO_DATE | |
| TO_TIMESTAMP | |
| UNIX_TIMESTAMP | |
| YEAR | |

Impala Numeric Function Support

TDV supports the numeric functions listed in the table below for Impala.

| Impala Numeric Function | Notes |
|-------------------------|-------|
| ABS | |
| ACOS | |
| ASIN | |
| ATAN | |
| CEIL | |
| CEILING | |

| Impala Numeric Function | Notes |
|-------------------------|--|
| COS | |
| DEGREES | |
| EXP | |
| FLOOR | |
| LN | |
| LOG | |
| PI | Returns the value of pi as a float value, with at least 6 digits of precision. Returned precision is greater without push. |
| POW | |
| POWER | |
| RANDOM, RAND | Every invocation returns a different value for TDV. Impala always returns the same value for RAND() if the number sequence is not changed for each query invocation. |
| RADIANS | |
| ROUND | <p>The value of the second parameter in Round(p,s) cannot exceed 38.</p> <p>There is different logic for round-off on the avg(distinct decimal) function.</p> <p>TDV does round-off, whereas Impala rounds-down.</p> |
| SIN | |
| SQRT | |
| TAN | |

Impala Push-Only Function Support

TDV supports the push-only functions listed in the table below for Impala.

| Impala Push-Only Function | Notes |
|---------------------------|--|
| FIND_IN_SET | |
| GET_JSON_OBJECT | |
| PARSE_URL | With two or three STRING arguments. |
| PERCENTILE | |
| PERCENTILE_APPROX | With two or three STRING arguments. |
| REGEXP | |
| REGEXP_EXTRACT | |
| REGEXP_REPLACE | |
| REVERSE | With STRING argument. |
| RLIKE, REGEXP_LIKE | The first parameter converted to upper case. One of the two parameters must be variable, not a literal. |
| TEST | |

Impala String Function Support

TDV supports the string functions listed in the table below for Impala.

| Impala String Function | Notes |
|------------------------|-------|
| ASCII | |
| CONCAT | |
| LCASE | |
| LENGTH | |
| LOWER | |

| Impala String Function | Notes |
|------------------------|---|
| LPAD | |
| LTRIM | |
| REPLACE | |
| RPAD | |
| RTRIM | |
| SPACE | |
| SUBSTR | Impala and TDV have different logic to handle substr(), when the start index is a negative value. |
| SUBSTRING | |
| TRIM | |
| UCASE | |
| UPPER | |

Informix Function Support

TDV supports the following types of functions for Informix:

- [Informix Aggregate Function Support, page 582](#)
- [Informix Character Function Support, page 582](#)
- [Informix Conditional Function Support, page 583](#)
- [Informix Conversion Function Support, page 583](#)
- [Informix Date Function Support, page 584](#)
- [Informix Numeric Function Support, page 584](#)

Informix Aggregate Function Support

TDV supports the aggregate functions listed in the table below for Informix.

| Informix Aggregate Function | Notes |
|-----------------------------|--|
| AVG | Not supported: <ul style="list-style-type: none">• BYTE arguments• TEXT arguments |
| COUNT | Not supported: <ul style="list-style-type: none">• BYTE arguments• COUNT (@VARBINARY)• COUNT (@LONGVARCHAR)• TEXT arguments |
| MAX | Not supported: <ul style="list-style-type: none">• BYTE arguments• MAX (@LONGVARCHAR).DISTINCT• TEXT arguments |
| MIN | Not supported: <ul style="list-style-type: none">• BYTE arguments• MIN (@LONGVARCHAR)• TEXT arguments |
| SUM | Not supported: <ul style="list-style-type: none">• BYTE arguments• TEXT arguments |

Informix Character Function Support

TDV supports the character functions listed in the table below for Informix.

| Informix Character Function | Notes |
|-----------------------------|-------|
| CONCAT | |

| Informix Character Function | Notes |
|-----------------------------|---------------------------------------|
| LENGTH | |
| LOWER | |
| REPLACE | |
| RTRIM | RTRIM (NULL) not allowed in Informix. |
| SUBSTRING | |
| TRIM | TRIM (NULL) not allowed in Informix. |
| UPPER | |

Informix Conditional Function Support

TDV supports the conditional function listed in the table below for Informix.

| Informix Conditional Function | Notes |
|-------------------------------|---|
| NULLIF | Not supported: <ul style="list-style-type: none"> LONGVARCHAR arguments VARBINARY arguments |

Informix Conversion Function Support

TDV supports the conversion functions listed in the table below for Informix.

| Informix Conversion Function | Notes |
|------------------------------|-------|
| CAST | |
| TO_CHAR | |
| TO_DATE | |
| TO_NUMBER | |

Informix Date Function Support

TDV supports the date functions listed in the table below for Informix.

| Informix Date Function | Notes |
|------------------------|-------|
| CURRENT_DATE | |
| DAY | |
| MONTH | |
| YEAR | |

Informix Numeric Function Support

TDV supports the numeric functions listed in the table below for Informix.

| Informix Numeric Function | Notes |
|---------------------------|----------------|
| ABS | |
| ACOS | |
| ASIN | |
| ATAN | |
| COS | |
| COT | Not supported. |
| EXP | |
| LOG | |
| POWER | |
| ROUND | |
| SIN | |
| SQRT | |
| TAN | |

JDBC Function Support

TDV supports the following type of function for JDBC:

- [JDBC Aggregate Function Support, page 585](#)

JDBC Aggregate Function Support

TDV supports the aggregate functions listed in the table below for JDBC.

| JDBC Aggregate Function | Notes |
|-------------------------|------------------------------------|
| AVG | Number arguments. |
| COUNT | All argument data types. |
| MAX | Number, string, or date arguments. |
| MIN | Number, string, or date arguments. |
| SUM | Number arguments. |

JSON Function Support

TDV supports the following functions for JSON.

| JSON Function | Notes |
|---------------|--|
| JSON_OBJECT | Evaluates a key-value pair and returns a JSON object containing the pair |
| JSON_ARRAY | Returns the listed values. The list can be empty. Array values must be of type string, number, object, array, boolean or null. |
| JSON_PATH | Is a query language with features similar to XPath that lets you extract just the bits of a JSON document your application needs. Array values must be of type string, number, object, array, boolean or null. |
| JSON_EXTRACT | The JSON_EXTRACT function can extract individual values from a JSON object |

| JSON Function | Notes |
|---------------|--|
| JSON_COUNT | Returns the number of elements in a JSON array within a JSON object. It returns the values based on the JSON path passed as the second argument to the function. |
| JSON_SUM | Returns the sum of the numeric values of a JSON array within a JSON object |
| JSON_MIN | Returns the lowest numeric value of a JSON array within a JSON object |
| JSON_MAX | Returns the highest numeric value of a JSON array within a JSON object |
| JSON_AVG | Returns the average value of a JSON array within a JSON object |

Microsoft Access Function Support

TDV supports the following types of functions for Microsoft Access:

- [Microsoft Access Aggregate Function Support, page 586](#)
- [Microsoft Access Analytic Aggregate Function Support, page 587](#)
- [Microsoft Access Character Function Support, page 587](#)
- [Microsoft Access Conditional Function Support, page 588](#)
- [Microsoft Access Conversion Function Support, page 588](#)
- [Microsoft Access Date Function Support, page 589](#)
- [Microsoft Access Numeric Function Support, page 589](#)

Microsoft Access Aggregate Function Support

TDV supports the aggregate functions listed in the table below for Microsoft Access.

Microsoft Access does not support the DISTINCT keyword in aggregate functions.

| Microsoft Access Aggregate Function | Notes |
|-------------------------------------|-------------------------|
| AVG | DISTINCT not supported. |
| COUNT | DISTINCT not supported. |
| MAX | DISTINCT has no effect. |
| MIN | DISTINCT has no effect. |
| SUM | DISTINCT not supported. |

Microsoft Access Analytic Aggregate Function Support

TDV supports the analytic aggregate functions listed in the table below for Microsoft Access.

| Microsoft Access Analytic Aggregate Function | Notes |
|--|-------|
| STDDEV_POP | |
| STDDEV_SAMP | |
| VARIANCE_POP | |
| VARIANCE_SAMP | |

Microsoft Access Character Function Support

TDV supports the character functions listed in the table below for Microsoft Access.

| Microsoft Access Character Function | Notes |
|-------------------------------------|-------|
| CONCAT | |
| LENGTH | |
| LOWER | |

| Microsoft Access Character Function | Notes |
|-------------------------------------|---|
| REPLACE | Even though Microsoft Access has this function, the driver does not recognize it. |
| RTRIM | |
| SPACE | |
| TRIM | |
| UPPER | |

Microsoft Access Conditional Function Support

TDV supports the conditional function listed in the table below for Microsoft Access.

| Function | Notes |
|----------|-------|
| NULLIF | |

Microsoft Access Conversion Function Support

TDV supports the conversion functions listed in the table below for Microsoft Access.

| Microsoft Access Conversion Function | Notes |
|--------------------------------------|--|
| CAST | |
| CDATE | NULL values can cause this function to fail. |
| CDBL | NULL values can cause this function to fail. |
| CSTR | NULL values can cause this function to fail. |
| TO_CHAR | |
| TO_DATE | |
| TO_NUMBER | |
| TO_TIMESTAMP | |

Microsoft Access Date Function Support

TDV supports the date functions listed in the table below for Microsoft Access.

| Microsoft Access Date Function | Notes |
|--------------------------------|-------|
| CURRENT_DATE | |
| CURRENT_TIME | |
| CURRENT_TIMESTAMP | |
| DAY | |
| MONTH | |
| YEAR | |

Microsoft Access Numeric Function Support

TDV supports the numeric functions listed in the table below for Microsoft Access.

| Microsoft Access Numeric Function | Notes |
|-----------------------------------|--|
| ABS | |
| ACOS | Not supported. |
| ASIN | Not supported. |
| ATAN | |
| COS | |
| COT | |
| EXP | |
| LOG | NULL values can cause this function to fail. |
| ROUND | |
| SIN | |
| TAN | |

Microsoft Excel Function Support

TDV supports the Microsoft Excel functions listed in the table below.

TDV Excel integration is through the Apache POI project. The following supported function list and limitations are based on the open source documentation for that project. Further details can be found on the Web.

| Function or Operator | Notes |
|----------------------|--|
| Operators | Arithmetic and logical operators; some region operators. |
| Built-in functions | More than 350 recognized. |
| Add-in functions | Three from Analysis Toolpak recognized. |

Limitations

The following is a list of some of the known limitations of TDV’s implementation of Microsoft Excel functions:

- TDV cannot manipulate Excel array or table formulas of the form “{=...}” (rather than of the form “=...”).
- TDV cannot handle the region operators (UNION and INTERSECTION).
- TDV cannot parse add-in functions that have not previously been called.
- TDV cannot preserve white space in formulas.
- TDV cannot convert charts or macros to TDV objects.
- TDV does not support pivot tables.

Microsoft SQL Server Function Support

TDV supports the following types of functions for Microsoft SQL Server:

- [Microsoft SQL Server Aggregate Function Support, page 591](#)
- [Microsoft SQL Server Analytic Function Support, page 591](#)
- [Microsoft SQL Server Analytic Aggregate Function Support, page 592](#)
- [Microsoft SQL Server Character Function Support, page 593](#)
- [Microsoft SQL Server Conditional Function Support, page 594](#)

- [Microsoft SQL Server Conversion Function Support, page 594](#)
- [Microsoft SQL Server Date Function Support, page 595](#)
- [Microsoft SQL Server Encryption Function Support, page 596](#)
- [Microsoft SQL Server Numeric Function Support, page 596](#)
- [Microsoft SQL Server Time Function Support, page 597](#)

Microsoft SQL Server Aggregate Function Support

TDV supports the aggregate functions listed in the table below for Microsoft SQL Server.

| Microsoft SQL Server Aggregate Function | Notes |
|---|-----------------------------------|
| AVG | Unique identifiers not supported. |
| COUNT | Unique identifiers not supported. |
| MAX | |
| MIN | |
| SUM | Unique identifiers not supported. |

Microsoft SQL Server Analytic Function Support

TDV supports the analytic functions listed in the table below for Microsoft SQL Server 2005 and 2008.

| Microsoft SQL Server Analytic Function | Notes |
|--|-------|
| AVG | |
| COUNT | |
| DENSE_RANK | |
| MAX | |
| MIN | |
| NTILE | |

| Microsoft SQL Server Analytic Function | Notes |
|--|-------|
| RANDOM | |
| RANK | |
| ROW_NUMBER | |
| STDDEV | |
| SUM | |
| VAR_POP | |
| VARIANCE | |

Microsoft SQL Server Analytic Aggregate Function Support

TDV supports the analytic aggregate functions listed in the table below for Microsoft SQL Server.

| Microsoft SQL Server Analytic Aggregate Function | Notes |
|--|---------------------------|
| CUM_DIST | SQL Server 2012 and 2014. |
| FIRST_VALUE | SQL Server 2012 and 2014. |
| LAG | SQL Server 2012 and 2014. |
| LAST_VALUE | SQL Server 2012 and 2014. |
| LEAD | SQL Server 2012 and 2014. |
| PERCENTILE_CONT | SQL Server 2012 and 2014. |
| PERCENTILE_DISC | SQL Server 2012 and 2014. |
| STDDEV | |
| STDDEV_POP | |
| STDDEV_SAMP | |
| VARIANCE | |
| VARIANCE_POP | |

| Microsoft SQL Server Analytic Aggregate Function | Notes |
|--|-------|
| VARIANCE_SAMP | |

Microsoft SQL Server Character Function Support

TDV supports the character functions listed in the table below for Microsoft SQL Server.

| Microsoft SQL Server Character Function | Notes |
|---|---|
| ASCII | <ul style="list-style-type: none"> |
| CONCAT | <ul style="list-style-type: none">Unique identifiers not supported.When the input timestamp value has no fractional seconds (hh:mm:ss), TDV does not print the fractional part. This is the way many data sources handle this situation. However, SQL Server <i>does</i> add the fractional part (hh:mm:ss.fff). |
| LENGTH | |
| LOWER | |
| POSITION | |
| REPLACE | |
| RTRIM | |
| SOUNDEX | |
| SPACE | |
| SUBSTRING | |
| TRIM | |
| UPPER | |

Microsoft SQL Server Conditional Function Support

TDV supports the conditional functions listed in the table below for Microsoft SQL Server.

| Microsoft SQL Server Conditional Function | Notes |
|---|--|
| COALESCE | |
| DECODE | Mapped to CASE. |
| ISNULL | |
| ISNUMERIC | |
| NULLIF | NULL literal cannot be the first argument to NULLIF function. NULLIF does not support IMAGE, NTEXT, or TEXT. |
| NVL | |

Microsoft SQL Server Conversion Function Support

TDV supports the conversion functions listed in the table below for Microsoft SQL Server. These conversion functions do not support unique identifiers.

| Microsoft SQL Server Conversion Function | Notes |
|--|---|
| CAST | <ul style="list-style-type: none">TINYINT is cast to SMALLINT.Cannot cast a number to DOUBLE.Cannot cast a string to any integer data type.Cannot cast floating point to DOUBLE. |
| FORMAT_DATE | |
| PARSE_TIMESTAMP | |
| TO_CHAR | |
| TO_DATE | |
| TO_NUMBER | |

| Microsoft SQL Server Conversion Function | Notes |
|--|-------|
| TO_TIMESTAMP | |

Microsoft SQL Server Date Function Support

TDV supports the date functions listed in the table below for Microsoft SQL Server.

| Microsoft SQL Server Date Function | Notes |
|------------------------------------|---|
| CURRENT_DATE | |
| CURRENT_TIMESTAMP | |
| DATEADD | SQL Server 2014 only. |
| DATEDIFF | SQL Server 2008, 2012, 2014. DATEDIFF (DATEPART, STARTDATE, ENDDATE) |
| DATEPART | SQL Server 2014 only. |
| DAY | |
| DAYNAME | SQL Server 2014 only. |
| DAYOFMONTH | SQL Server 2014 only. |
| DAYOFWEEK | SQL Server 2014 only. |
| HOURL | SQL Server 2014 only. |
| MINUTE | SQL Server 2014 only. |
| MONTH | |
| MONTHNAME | SQL Server 2014 only. |
| QUARTER | SQL Server 2014 only. |
| SECOND | SQL Server 2014 only. |
| WEEK | SQL Server 2014 only. |
| YEAR | |

Microsoft SQL Server Encryption Function Support

TDV supports the encryption functions listed in the table below for Microsoft SQL Server 2008. If the SQL Server string data type is not CHAR or VARCHAR, the results are different when function is not pushed.

| Microsoft SQL Server Encryption Function | Notes |
|--|-------|
| HASHMD2 | |
| HASHMD5 | |
| HASHSHA | |
| HASHSHA1 | |

Microsoft SQL Server Numeric Function Support

TDV supports the numeric functions listed in the table below for Microsoft SQL Server.

Notes:

- These numeric functions do not support unique identifiers.
- Microsoft SQL Server 2005, 2008, and 2012 support floating point and numeric data types for both arguments of a modulo (%) operator.

| Microsoft SQL Server Numeric Function | Notes |
|---------------------------------------|-------|
| ABS | |
| ACOS | |
| ASIN | |
| ATAN | |
| CEILING | |
| COS | |
| COT | |
| DEGREES | |
| EXP | |

| Microsoft SQL Server Numeric Function | Notes |
|---------------------------------------|-------|
| FLOOR | |
| LOG | |
| PI | |
| POWER | |
| RADIANS | |
| ROUND | |
| SIN | |
| SQRT | |
| TAN | |

Microsoft SQL Server Time Function Support

TDV supports the time function listed in the table below for Microsoft SQL Server.

| Function | Notes |
|----------|-------|
| EXTRACT | |

MySQL Function Support

TDV supports the following types of functions for MySQL:

- [MySQL Aggregate Function Support, page 598](#)
- [MySQL Analytic Function Support, page 598](#)
- [MySQL Analytic Aggregate Function Support, page 598](#)
- [MySQL Character Function Support, page 599](#)
- [MySQL Conditional Function Support, page 599](#)
- [MySQL Conversion Function Support, page 600](#)
- [MySQL Date Function Support, page 601](#)

- [MySQL Numeric Function Support, page 601](#)
- [MySQL Time Function Support, page 602](#)

Note: If MySQL returns data instead of an error message when, for example, “xk” is CAST as an INTEGER, set SQL_MODE to TRADITIONAL in the MySQL database. This makes function results the same for push and no-push.

MySQL Aggregate Function Support

TDV supports the aggregate functions listed in the table below for MySQL.

| MySQL Aggregate Function | Notes |
|--------------------------|--|
| AVG | DISTINCT and STRING are not supported. |
| COUNT | DISTINCT not supported. |
| MAX | DISTINCT not supported. |
| MIN | DISTINCT not supported. |
| SUM | DISTINCT not supported. |

MySQL Analytic Function Support

TDV supports the analytic function listed in the table below for MySQL 3.0.

| MySQL Analytic Function | Notes |
|-------------------------|-------|
| STDDEV_POP | |

MySQL Analytic Aggregate Function Support

TDV supports the analytic aggregate functions listed in the table below for MySQL.

| MySQL Analytic Aggregate Function | Notes |
|-----------------------------------|-------------------|
| STDDEV | MySQL 3.0 only. |
| STDDEV_SAMP | MySQL 5.0.3 only. |
| VAR_SAMP | MySQL 5.0.3 only. |

| MySQL Analytic Aggregate Function | Notes |
|-----------------------------------|-------------------------|
| VARIANCE_POP | MySQL 4.1 and 5.0 only. |

MySQL Character Function Support

TDV supports the character functions listed in the table below for MySQL.

| MySQL Character Function | Notes |
|--------------------------|--|
| CONCAT | |
| LENGTH | |
| LOWER | |
| POSITION | Case-sensitive in MySQL 3.23. In MySQL 4.0 and later, case-sensitive only if the arguments are binary strings. |
| REPLACE | |
| RTRIM | |
| SPACE | |
| SUBSTRING | |
| TRIM | |
| UPPER | |

MySQL Conditional Function Support

TDV supports the conditional functions listed in the table below for MySQL.

| MySQL Conditional Function | Notes |
|----------------------------|-----------------|
| COALESCE | |
| DECODE | Mapped to CASE. |

MySQL Conversion Function Support

TDV supports the conversion functions listed in the table below for MySQL. Conversion functions map data types differently depending on the MySQL version.

| MySQL Conversion Function | Notes |
|---------------------------|---|
| CAST | <ul style="list-style-type: none">Supported only for MySQL 4.0.2 or higher.If a JConnector version prior to 5.1.28 is used, fractional seconds returned from CAST STRING to TIMESTAMP are erroneously offset three decimal places to the right.Casting as a whole number converts NULL values to zero, so whole numbers cannot be pushed safely.When casting a NULL value for a TIMESTAMP column as VARCHAR, MySQL might return either NULL or '0000-00-00 00:00:00' (depending on the default-value setting for the column in the data source). TDV server (no push) always returns NULL.Depending on MySQL server-side settings (TRADITIONAL vs. STRICT), can return data and warnings or no data and error when casting incompatible data types. |
| FORMAT_DATE | Supported only for MySQL 4.1.1 or higher. |
| PARSE_DATE | Supported only for MySQL 4.1.1 or higher. |
| PARSE_TIME | Supported only for MySQL 4.1.1 or higher. |
| PARSE_TIMESTAMP | Supported only for MySQL 4.1.1 or higher. |
| TO_CHAR | Supported only for MySQL 4.0.2 or higher. |
| TO_DATE | Supported only for MySQL 4.0.2 or higher. Variant mappings for different versions. |
| TO_NUMBER | Supported only for MySQL 4.0.2 or higher. |
| TO_TIMESTAMP | Supported only for MySQL 4.0.2 or higher. |

MySQL Date Function Support

TDV supports the date functions listed in the table below for MySQL.

| MySQL Date Function | Notes |
|---------------------|-------------------------------|
| CURRENT_DATE | |
| CURRENT_TIME | |
| CURRENT_TIMESTAMP | |
| DATEDIFF | DATEDIFF (enddate, startdate) |
| DAY | |
| MONTH | |
| UTC_TO_TIMESTAMP | |
| YEAR | |

MySQL Numeric Function Support

TDV supports the numeric functions listed in the table below for MySQL.

| MySQL Numeric Function | Notes |
|------------------------|-------|
| ABS | |
| ACOS | |
| ASIN | |
| ATAN | |
| CEILING | |
| COS | |
| COT | |
| DEGREES | |
| EXP | |
| FLOOR | |

| MySQL Numeric Function | Notes |
|------------------------|---|
| LOG | MySQL 5.5 returns natural log (base e) of a number. TDV server default is base 10. |
| POWER | |
| RADIANS | |
| RANDOM | |
| ROUND | |
| SIN | |
| SQRT | |
| TAN | |

MySQL Time Function Support

TDV supports the time function listed in the table below for MySQL versions 4.1 and 5.0.

| MySQL Time Function | Notes |
|---------------------|-------|
| EXTRACT | |

NeoView Function Support

- NULL is not allowed as input to any NeoView functions.
- TDV supports the following types of functions for Neoview:
- [NeoView Aggregate Function Support, page 603](#)
 - [NeoView Character Function Support, page 603](#)
 - [NeoView Conditional Function Support, page 604](#)
 - [NeoView Conversion Function Support, page 604](#)
 - [NeoView Date Function Support, page 604](#)
 - [NeoView Numeric Function Support, page 605](#)

NeoView Aggregate Function Support

TDV supports the aggregate functions listed in the table below for Neoview.

| NeoView Aggregate Function | Notes |
|----------------------------|---|
| AVG | Numeric arguments only; STRING not supported. |
| COUNT | |
| MAX | DISTINCT is pushed. |
| MIN | DISTINCT is pushed. |
| SUM | Numeric arguments only; STRING not supported. |

NeoView Character Function Support

TDV supports the character functions listed in the table below for Neoview.

| NeoView Character Function | Notes |
|----------------------------|-------|
| CONCAT | |
| LENGTH | |
| LOWER | |
| POSITION | |
| REPLACE | |
| RTRIM | |
| SUBSTRING | |
| TRIM | |
| UPPER | |

NeoView Conditional Function Support

TDV supports the conditional functions listed in the table below for Neoview.

| NeoView Conditional Function | Notes |
|------------------------------|-------|
| DECODE | |
| NULLIF | |

NeoView Conversion Function Support

TDV supports the conversion functions listed in the table below for Neoview.

| NeoView Conversion Function | Notes |
|-----------------------------|-------|
| CAST | |
| TO_CHAR | |
| TO_DATE | |
| TO_NUMBER | |
| TO_TIMESTAMP | |

NeoView Date Function Support

TDV supports the date functions listed in the table below for Neoview.

| NeoView Date Function | Notes |
|-----------------------|-------|
| CURRENT_DATE | |
| CURRENT_TIME | |
| CURRENT_TIMESTAMP | |
| DAY | |
| MONTH | |
| YEAR | |

NeoView Numeric Function Support

TDV supports the numeric functions listed in the table below for Neoview.

| NeoView Numeric Function | Notes |
|--------------------------|-------|
| ABS | |
| ACOS | |
| ASIN | |
| ATAN | |
| CEILING | |
| COS | |
| COT | |
| DEGREES | |
| EXP | |
| FLOOR | |
| LOG | |
| PI | |
| POWER | |
| RADIANS | |
| ROUND | |
| SIN | |
| SQRT | |
| TAN | |

Netezza Function Support

TDV supports the following types of functions for Netezza:

- [Netezza Aggregate Function Support, page 606](#)
- [Netezza Analytic Function Support, page 608](#)
- [Netezza Analytic Aggregate Function Support, page 609](#)
- [Netezza Binary Function Support, page 610](#)
- [Netezza Character Function Support, page 610](#)
- [Netezza Conditional Function Support, page 612](#)
- [Netezza Conversion Function Support, page 613](#)
- [Netezza Date Function Support, page 613](#)
- [Netezza Numeric Function Support, page 614](#)
- [Netezza Phonetic Function Support, page 617](#)
- [Netezza Statistical Analytic Aggregate Function Support, page 617](#)
- [Netezza Time Function Support, page 618](#)

Netezza Aggregate Function Support

TDV supports the aggregate functions listed in the table below for Netezza.

| Netezza Aggregate Function | Notes |
|----------------------------|---|
| AVG | Not supported: <ul style="list-style-type: none">• AVG (BOOLEAN)• AVG (BOOLEAN) DISTINCT• AVG (NCHAR)• AVG (NCHAR) DISTINCT• AVG (NVARCHAR)• AVG (NVARCHAR) DISTINCT |
| COUNT | |

| Netezza Aggregate Function | Notes |
|----------------------------|---|
| MAX | <p>Not supported:</p> <ul style="list-style-type: none"> • MAX (BOOLEAN) • MAX (BOOLEAN) DISTINCT • MAX (NCHAR) • MAX (NCHAR) DISTINCT • MAX (NVARCHAR) • MAX (NVARCHAR) DISTINCT <p>In version 4.5, 5.0, 6.0: analytic, with the same arguments not supported. MAX(NULL) analytic is NULL.</p> |
| MIN | <p>Not supported:</p> <ul style="list-style-type: none"> • MIN (BOOLEAN) • MIN (BOOLEAN) DISTINCT • MIN (NCHAR) • MIN (NCHAR) DISTINCT • MIN (NVARCHAR) • MIN (NVARCHAR) DISTINCT |
| SUM | <p>Not supported:</p> <ul style="list-style-type: none"> • SUM (BOOLEAN) • SUM (BOOLEAN).DISTINCT • SUM (NCHAR) • SUM (NCHAR).DISTINCT • SUM (NVARCHAR) • SUM (NVARCHAR).DISTINCT |

Netezza Analytic Function Support

TDV supports the analytic functions listed in the table below for Netezza.

| Netezza Analytic Function | Notes |
|---------------------------|------------------------|
| DENSE_RANK | Version 4.5, 5.0, 6.0. |
| FIRST_VALUE | Version 4.5, 5.0, 6.0. |
| LAG | Version 4.5, 5.0, 6.0. |
| LAST_VALUE | Version 4.5, 5.0, 6.0. |
| LEAD | Version 4.5, 5.0, 6.0. |
| RANK | Version 4.5, 5.0, 6.0. |
| ROW_NUMBER | Version 4.5, 5.0, 6.0. |
| STDDEV | Version 4.5. |
| STDDEV_POP | Version 4.5. |
| STDDEV_SAMP | Version 4.5. |
| VAR_POP | Version 4.5. |
| VAR_SAMP | Version 4.5. |
| VARIANCE | Version 4.5. |
| VARIANCE_POP | Version 4.5. |
| VARIANCE_SAMP | Version 4.5. |

Netezza Analytic Aggregate Function Support

TDV supports the analytic aggregate functions listed in the table below for Netezza versions 4.5, 5.0, and 6.0.

| Netezza Analytic Aggregate Function | Notes |
|-------------------------------------|---|
| AVG | <p>AVG (NULL) is NULL.</p> <p>Not supported:</p> <ul style="list-style-type: none">• AVG (BOOLEAN)• AVG (NCHAR)• AVG (NCHAR) DISTINCT• AVG (NVARCHAR)• AVG (NVARCHAR) DISTINCT |
| COUNT | |
| MAX | <p>MAX (NULL) is NULL.</p> <p>Not supported:</p> <ul style="list-style-type: none">• MAX (BOOLEAN)• MAX (BOOLEAN) DISTINCT• MAX (NCHAR)• MAX (NCHAR) DISTINCT• MAX (NVARCHAR)• MAX (NVARCHAR) DISTINCT |
| MIN | <p>MIN (NULL) is NULL.</p> <p>Not supported:</p> <ul style="list-style-type: none">• MIN (BOOLEAN)• MIN (BOOLEAN) DISTINCT• MIN (NCHAR)• MIN (NCHAR) DISTINCT• MIN (NVARCHAR)• MIN (NVARCHAR) DISTINCT |

| Netezza Analytic Aggregate Function | Notes |
|-------------------------------------|--|
| SUM | SUM (NULL) is NULL. Not supported: <ul style="list-style-type: none">SUM (BOOLEAN)SUM (BOOLEAN) DISTINCTSUM (NCHAR)SUM (NCHAR) DISTINCTSUM (NVARCHAR)SUM (NVARCHAR) DISTINCT |

Netezza Binary Function Support

TDV supports the binary functions listed in the table below for Netezza versions 5.0 and 6.0.

| Netezza Binary Function | Notes |
|------------------------------------|----------------------|
| INT1AND, INT2AND, INT4AND, INT8AND | Bitwise AND |
| INT1NOT, INT2NOT, INT4NOT, INT8NOT | Bitwise NOT |
| INT1OR, INT2OR, INT4OR, INT8OR | Bitwise OR |
| INT1SHL, INT2SHL, INT4SHL, INT8SHL | Bitwise shift left |
| INT1SHR, INT2SHR, INT4SHR, INT8SHR | Bitwise shift right |
| INT1XOR, INT2XOR, INT4XOR, INT8XOR | Bitwise EXCLUSIVE OR |

Netezza Character Function Support

TDV supports the character functions listed in the table below for Netezza.

| Netezza Character Function | Notes |
|----------------------------|-----------------------|
| ASCII | Versions 5.0 and 6.0. |
| BTRIM | Versions 5.0 and 6.0. |

| Netezza Character Function | Notes |
|----------------------------|--|
| CHR | Versions 5.0 and 6.0. |
| CONCAT | Not supported: <ul style="list-style-type: none"> • BIT argument • NCHAR argument • NVARCHAR argument |
| DLE_DST | Versions 5.0 and 6.0. |
| INITCAP | Versions 5.0 and 6.0. |
| INSTR | Versions 5.0 and 6.0. |
| LE_DST | Versions 5.0 and 6.0. |
| LENGTH | |
| LOWER | |
| LPAD | Versions 5.0 and 6.0. Length limit of 4000. |
| LTRIM | Versions 5.0 and 6.0. |
| POSITION | Not supported: <ul style="list-style-type: none"> • NCHAR argument • NVARCHAR argument |
| REPEAT | Versions 5.0 and 6.0. |
| REPLACE | Not available in Netezza. Netezza has a TRANSLATE function, but it works differently. |
| RPAD | Versions 5.0 and 6.0. |
| RTRIM | Versions 5.0 and 6.0. |
| SOUNDEX | |
| SPACE | Not supported: <ul style="list-style-type: none"> • BIT argument |
| STRPOS | Versions 5.0 and 6.0. |

| Netezza Character Function | Notes |
|----------------------------|---|
| SUBSTR | Versions 5.0 and 6.0. |
| SUBSTRING | Not supported: <ul style="list-style-type: none">NCHAR argumentNVARCHAR argument |
| TRANSLATE | Versions 5.0 and 6.0. |
| TRIM | Versions 5.0 and 6.0. |
| TRUNC | Versions 5.0 and 6.0. |
| UNICHR | Versions 5.0 and 6.0. |
| UNICODE | Versions 5.0 and 6.0. |
| UPPER | |

Netezza Conditional Function Support

TDV supports the conditional functions listed in the table below for Netezza.

| Netezza Conditional Function | Notes |
|------------------------------|--|
| COALESCE | |
| DECODE | Versions 5.0 and 6.0. |
| NULLIF | Not supported: <ul style="list-style-type: none">BIT argumentINTERVAL argumentNCHAR argumentNVARCHAR argument |

Netezza Conversion Function Support

TDV supports the conversion functions listed in the table below for Netezza.

| Netezza Conversion Function | Notes |
|-----------------------------|---|
| CAST | Not supported: <ul style="list-style-type: none"> • BIT argument • NCHAR first argument • NULL first argument • NVARCHAR first argument • <any_number> AS NULL |
| TO_CHAR | Not supported: <ul style="list-style-type: none"> • BIT argument • NCHAR argument • NVARCHAR argument |
| TO_DATE | Not supported: <ul style="list-style-type: none"> • NCHAR argument • NVARCHAR argument |
| TO_NUMBER | Not supported: <ul style="list-style-type: none"> • BIT argument • NCHAR argument • NVARCHAR argument |
| TO_TIMESTAMP | Not supported: <ul style="list-style-type: none"> • NCHAR argument • NVARCHAR argument |

Netezza Date Function Support

TDV supports the date functions listed in the table below for Netezza.

| Netezza Date Function | Notes |
|-----------------------|-------|
| CURRENT_DATE | |

| Netezza Date Function | Notes |
|-----------------------|-------|
| CURRENT_TIME | |
| CURRENT_TIMESTAMP | |
| DAY | |
| MONTH | |
| YEAR | |

Netezza Numeric Function Support

TDV supports the numeric functions listed in the table below for Netezza.

| Netezza Numeric Function | Notes |
|--------------------------|--|
| ABS | Not supported: <ul style="list-style-type: none">• BIT argument• NCHAR argument• NVARCHAR argument |
| ACOS | Not supported: <ul style="list-style-type: none">• BIT argument• NCHAR argument• NVARCHAR argument |
| ASIN | Not supported: <ul style="list-style-type: none">• BIT argument• NCHAR argument NVARCHAR argument |
| ATAN | Not supported: <ul style="list-style-type: none">• BIT argument• NCHAR argument• NVARCHAR argument |
| ATAN2 | Versions 5.0 and 6.0. |

| Netezza Numeric Function | Notes |
|--------------------------|--|
| CEIL | Versions 5.0 and 6.0. |
| CEILING | |
| COS | Not supported: <ul style="list-style-type: none"> • BIT argument • NCHAR argument • NVARCHAR argument |
| COT | Not supported: <ul style="list-style-type: none"> • BIT argument • NCHAR argument • NVARCHAR argument |
| DEGREES | |
| EXP | Not supported: <ul style="list-style-type: none"> • BIT argument • NCHAR argument • NVARCHAR argument |
| FACTORIAL | Versions 5.0 and 6.0. |
| FLOOR | |
| LN | Versions 5.0 and 6.0. |
| LOG | Not supported: <ul style="list-style-type: none"> • BIT argument • NCHAR argument • NVARCHAR argument |
| NVL | |
| NVL2 | |
| PI | |
| POW | Versions 5.0 and 6.0. |

| Netezza Numeric Function | Notes |
|--------------------------|--|
| POWER | Not supported: <ul style="list-style-type: none">• BIT argument• NCHAR argument• NVARCHAR argument |
| RADIANS | |
| RANDOM | Versions 5.0 and 6.0. |
| ROUND | Not supported: <ul style="list-style-type: none">• BIT argument• NCHAR argument• NVARCHAR argument |
| SIGN | |
| SIN | Not supported: <ul style="list-style-type: none">• BIT argument• NCHAR argument• NVARCHAR argument |
| SQRT | Not supported: <ul style="list-style-type: none">• BIT argument• NCHAR argument• NVARCHAR argument |
| TAN | Not supported: <ul style="list-style-type: none">• BIT argument• NCHAR argument• NVARCHAR argument |

Netezza Phonetic Function Support

TDV supports the phonetic functions listed in the table below for Netezza versions 5.0 and 6.0.

| Netezza Phonetic Function | Notes |
|---------------------------|-------|
| DBL_MP | |
| NYSIIS | |
| PRI_MP | |
| SCORE_MP | |
| SEC_MP | |

Netezza Statistical Analytic Aggregate Function Support

TDV supports the statistical analytic aggregate functions listed in the table below for Netezza version 4.5.

| Netezza Statistical Analytic Aggregate Function | Notes |
|---|-------|
| STDDEV | |
| STDDEV_POP | |
| STDDEV_SAMP | |
| VAR_POP | |
| VAR_SAMP | |
| VARIANCE | |
| VARIANCE_POP | |
| VARIANCE_SAMP | |

Netezza Time Function Support

TDV supports the time functions listed in the table below for Netezza.

| Netezza Time Function | Notes |
|-----------------------|-----------------------|
| ADD_MONTHS | Versions 5.0 and 6.0. |
| DATE_PART | Versions 5.0 and 6.0. |
| DATE_TRUNC | Versions 5.0 and 6.0. |
| EXTRACT | |
| LAST_DAY | Versions 5.0 and 6.0. |
| MONTHS_BETWEEN | Versions 5.0 and 6.0. |
| NEXT_DAY | Versions 5.0 and 6.0. |
| NOW | Versions 5.0 and 6.0. |
| TIMEOFDAY | Versions 5.0 and 6.0. |
| TIMESTAMP | Versions 5.0 and 6.0. |

Oracle Function Support

TDV has made every effort to support all of the aggregate and analytic functions that Oracle supports. The following table lists the Oracle functions and notes describing how TDV interprets the functions.

Aggregate functions return a single result row based on groups of rows, rather than based on single rows. Aggregate functions can appear in SELECT lists and in ORDER BY and HAVING clauses. They are commonly used with the GROUP BY clause in a SELECT statement, where Oracle Database divides the rows of a queried table or view into groups.

TDV supports the following types of functions for Oracle:

- [Oracle Aggregate Function Support, page 619](#)
- [Oracle Analytic Function Support, page 620](#)
- [Oracle Analytic Aggregate Function Support, page 621](#)
- [Oracle Binary Function Support, page 622](#)

- [Oracle Character Function Support, page 622](#)
- [Oracle Conditional Function Support, page 623](#)
- [Oracle Conversion Function Support, page 624](#)
- [Oracle Date Function Support, page 625](#)
- [Oracle Encryption Function Support, page 625](#)
- [Oracle Numeric Function Support, page 626](#)
- [Oracle Time Function Support, page 627](#)
- [Oracle XML Function Support, page 627](#)

Oracle Aggregate Function Support

TDV supports the aggregate functions listed in the table below for Oracle.

| Oracle Aggregate Function | Notes |
|---------------------------|--|
| AVG | Does not support whole numbers, because Oracle returns floating-point numbers instead of integers in the result. |
| CORR | Versions 9i, 10g, and 11g. |
| COUNT | BLOB and CLOB not supported. In 9i, 10g, and 11g, also not supported: BFILE, LONG, LONG RAW, LONGVARCHAR, NCLOB, or VARBINARY. |
| COVAR_POP | Versions 9i, 10g, and 11g. |
| COVAR_SAMP | Versions 9i, 10g, and 11g. |
| factorial sign (!) | Argument can be any whole number. |
| LISTAGG | Version 11g. |
| MAX | |
| MIN | |
| PERCENTILE_CONT | Versions 9i, 10g, and 11g. |
| PERCENTILE_DISC | Versions 9i, 10g, and 11g. |
| SUM | |

Oracle Analytic Function Support

TDV supports the analytic functions listed in the table below for Oracle.

Analytic functions are commonly used in data warehousing environments. They are all push-only functions.

| Oracle Analytic Function | Notes |
|--------------------------|---|
| AVG | Version 9i. |
| CORR | Versions 9i, 10g, and 11g. |
| COUNT | Version 9i. |
| COVAR_POP | Versions 9i, 10g, and 11g. |
| COVAR_SAMP | Versions 9i, 10g, and 11g. |
| CUME_DIST | Versions 9i, 10g, and 11g. |
| DENSE_RANK | Versions 9i, 10g, and 11g. |
| FIRST_VALUE | Versions 9i, 10g, and 11g. |
| LAG | Versions 9i, 10g, and 11g. |
| LAST_VALUE | Versions 9i, 10g, and 11g. |
| LEAD | Versions 9i, 10g, and 11g. |
| LISTAGG | Version 11g R2. Pushed. (Aggregate LISTAGG function is supported in TDV. (See LISTAGG , page 532 .) |
| MAX | Version 9i. |
| MEDIAN | Versions 9i, 10g, and 11g. MEDIAN (DISTINCT) not supported. |
| MIN | Version 9i. |
| NTILE | Versions 9i, 10g, and 11g. |
| PERCENT_RANK | Versions 9i, 10g, and 11g. |
| PERCENTILE_CONT | Versions 9i, 10g, and 11g. |

| Oracle Analytic Function | Notes |
|--------------------------|----------------------------|
| PERCENTILE_DISC | Versions 9i, 10g, and 11g. |
| RANK | Versions 9i, 10g, and 11g. |
| RATIO_TO_REPORT | Versions 9i, 10g, and 11g. |
| ROW_NUMBER | Versions 9i, 10g, and 11g. |
| SUM | Version 9i. |

Oracle Analytic Aggregate Function Support

TDV supports the analytic aggregate functions listed in the table below for Oracle.

Analytic functions are commonly used in data warehousing environments. They are all push-only functions.

| Oracle Analytic Aggregate Function | Notes |
|------------------------------------|----------------------------|
| LAST_VALUE | Versions 9i, 10g, and 11g. |
| PERCENTILE_CONT | Versions 9i, 10g, and 11g. |
| PERCENTILE_DISC | Versions 9i, 10g, and 11g. |
| STDDEV | |
| STDDEV_POP | |
| STDDEV_SAMP | |
| VAR_POP | Same as VARIANCE_POP. |
| VAR_SAMP | Same as VARIANCE_SAMP. |
| VARIANCE | Versions 9i. |
| VARIANCE_POP | |
| VARIANCE_SAMP | |

Oracle Binary Function Support

TDV supports the binary functions listed in the table below for Oracle 9i, 10g, and 11g.

| Oracle Binary Function | Notes |
|------------------------|-------|
| INT1AND | |
| INT2AND | |
| INT4AND | |
| INT8AND | |

Oracle Character Function Support

TDV supports the character functions listed in the table below for Oracle.

| Oracle Character Function | Notes |
|---------------------------|----------------------------|
| ASCII | |
| BTRIM | |
| CHR | |
| CONCAT | |
| GREATEST | Version 11g. |
| INITCAP | |
| INSTR | Case-sensitive by default. |
| LEAST | Version 11g. |
| LENGTH | |
| LOWER | |
| LPAD | |
| LTRIM | |

| Oracle Character Function | Notes |
|---------------------------|--|
| POSITION | Follows SQL-92 (STRICT). Not supported: mixing string, number, or date with NCHAR, NVARCHAR, or NVARCHAR2. |
| REPLACE | |
| RPAD | |
| RTRIM | |
| SOUNDEX | Returns a phonetic representation of a string. |
| SPACE | BIT not supported. Oracle returns SPACE(0) as NULL, but SQL-92 calls for ' '. |
| STRPOS | |
| SUBSTR | |
| SUBSTRING | Oracle does not follow SQL-92 standard. STRICT forces use of TDV, which follows the standard. |
| TRANSLATE | |
| TRIM | |
| UNICHR | |
| UPPER | |

Oracle Conditional Function Support

TDV supports the conditional functions listed in the table below for Oracle.

| Oracle Conditional Function | Notes |
|-----------------------------|---|
| COALESCE | Versions 9i, 10g, and 11g. |
| DECODE | |
| NULLIF | Versions 9i, 10g, and 11g. NULL cannot be the first argument. Does not support BFILE, BLOB, CLOB, LONG, or LONGVARCHAR. |

| Oracle Conditional Function | Notes |
|-----------------------------|-------|
| NVL | |
| NVL2 | |

Oracle Conversion Function Support

TDV supports the conversion functions listed in the table below for Oracle.

| Oracle Conversion Function | Notes |
|----------------------------|--|
| CAST | |
| FORMAT_DATE | For timestamps, Oracle omits fractional parts unless the format string “ff” is used. If “ff” does not specify a precision, Oracle returns the precision of the data type used to store the returned value. |
| PARSE_DATE | Does not push. |
| PARSE_TIMESTAMP | Format “ff” (fractional part) is valid only for TO_TIMESTAMP. |
| TO_CHAR | For timestamps, Oracle omits fractional parts unless the format string “ff” is used. If “ff” does not specify a precision, Oracle returns the precision of the data type used to store the returned value. |
| TO_DATE | Format “ff” (fractional part) does not work for this function. |
| TO_NUMBER | |
| TO_TIMESTAMP | For timestamps, Oracle omits fractional parts unless the format string “ff” is used. If “ff” does not specify a precision, Oracle returns the precision of the data type used to store the returned value. |

Oracle Date Function Support

TDV supports the date functions listed in the table below for Oracle.

| Oracle Date Function | Notes |
|----------------------|----------------------------|
| ADD_MONTHS | |
| CURRENT_DATE | |
| CURRENT_TIME | Not supported. |
| CURRENT_TIMESTAMP | Versions 9i, 10g, and 11g. |
| DATE_TRUNC | |
| DAY | |
| DAYS_BETWEEN | TDV/Studio 5.2 and later. |
| LAST_DAY | |
| MONTH | |
| MONTHS_BETWEEN | |
| NEXT_DAY | |
| TRUNC | |
| YEAR | |

Oracle Encryption Function Support

TDV supports the encryption functions listed in the table below for Oracle version 10g with DBMS_CRYPTO package.

| Oracle Encryption Function | Notes |
|----------------------------|-------|
| HASHMD5 | |
| HASHSHA1 | |

Oracle Numeric Function Support

TDV supports the numeric functions listed in the table below for Oracle.

| Oracle Numeric Function | Notes |
|-------------------------|---|
| ABS | |
| ACOS | |
| ASIN | |
| ATAN | |
| ATAN2 | |
| CEIL | |
| CEILING | |
| COS | |
| COT | |
| DEGREES | Not available. |
| DENSE_RANK | Versions 9i, 10g, and 11g. |
| EXP | |
| FLOOR | |
| LN | BIT, NCHAR, and NVARCHAR not supported. |
| LOG | |
| MEDIAN | Versions 9i, 10g, and 11g. MEDIAN (DISTINCT) not supported. |
| MOD | |
| PI | Not available. |
| POW | Not supported: mixing string, number, or NULL with BIT, NCHAR, or NVARCHAR. |
| POWER | |

| Oracle Numeric Function | Notes |
|-------------------------|----------------|
| RADIANS | Not available. |
| RANDOM | |
| ROUND | |
| SIGN | |
| SIN | |
| SQRT | |
| TAN | |
| TRUNC | |
| VARIANCE | |

Oracle Time Function Support

TDV supports the time functions listed in the table below for Oracle.

| Oracle Time Function | Notes |
|----------------------|-------|
| EXTRACT | |
| NOW | |
| TIMEOFDAY | |

Oracle XML Function Support

TDV supports the XML functions listed in the table below for Oracle.

| Oracle XML Function | Notes |
|---------------------|--|
| XMLAGG | Versions 9i, 10g, and 11g. |
| XMLATTRIBUTES | Versions 9i, 10g, and 11g. |
| XMLCOMMENT | Not supported, because it returns the following error: ORA-00932: inconsistent data types: expected - got CHAR. |
| XMLCONCAT | Versions 9i, 10g, and 11g. |

| Oracle XML Function | Notes |
|---------------------|--|
| XMLDOCUMENT | Oracle does not support this function. |
| XMLELEMENT | Versions 9i, 10g, and 11g. |
| XMLFOREST | Versions 9i, 10g, and 11g. |
| XMLTEXT | Oracle does not support this function. |

ParStream Function Support

ParStream does not support multi-value, string and date type as operand.

| Function type | ParStream Function | TDV Function |
|----------------------|-------------------------------------|---------------|
| Aggregate Function | AVG | AVG |
| | COUNT | COUNT |
| | MAX | MAX |
| | MIN | MIN |
| | SUM | SUM |
| | TAKE (columnname) | Not Supported |
| | STDDEV_POP | STDDEV_POP |
| Bit Function | BIT | Not Supported |
| Character Function | LOWER (value) | LOWER |
| | LOWERCASE (value) | LOWER |
| | UPPER (value) | UPPER |
| | UPPERCASE (value) | UPPER |
| Conditional Function | COALESCE | COALESCE |
| | IF (value, trueresult, falseresult) | Not Supported |
| | IFNULL (value, replacement) | IFNULL |
| Convert Function | CAST (val AS type) | CAST |

| Function type | ParStream Function | TDV Function |
|---------------|----------------------------|---|
| | TRUNC (val) | TRUNC |
| Date Function | DATE_PART (partname, col) | DATE_PART |
| | | The following combinations are not supported: |
| | | date_part('day', TIME) |
| | | date_part('month', TIME) |
| | | date_part('year', TIME) |
| | | date_part('dow', TIME) |
| | | date_part('doy', TIME) |
| | | date_part('epoch', TIME) |
| | | date_part('quarter', TIME) |
| | | date_part('week', TIME) |
| | | date_part('hour', DATE) |
| | | date_part('minute', DATE) |
| | | date_part('second', DATE) |
| | | date_part('millisecond', DATE) |
| | DATE_TRUNC (truncval, col) | DATE_TRUNC |
| | DAYOFMONTH (column) | DAYOFMONTH |
| | DAYOFWEEK (column) | DAYOFWEEK |
| | DAYOFYEAR (column) | DAYOFYEAR |
| | EPOCH (column) | EXTRACTEPOCH |
| | EXTRACT (part FROM column) | EXTRACT |
| | HOUR (column) | HOUR |
| | MILLISECOND (column) | EXTRACTMILLISECOND |

| Function type | ParStream Function | TDV Function |
|--------------------|--------------------|----------------|
| | MINUTE (column) | EXTRACTMINUTE |
| | QUARTER (column) | EXTRACTQUARTER |
| | SECOND (column) | EXTRACTSECOND |
| | WEEK (column) | EXTRACTWEEK |
| | MONTH (column) | EXTRACTMONTH |
| | YEAR (column) | EXTRACTYEAR |
| Numeric Function | FLOOR (val) | FLOOR |
| | MOD | MOD |
| Push Only Function | FIRST (value) | Not Supported |
| | DISTVALUES (col) | Not Supported |
| | HASH64 (strvalue) | Not Supported |

PostgreSQL Function Support

- TDV supports the following types of functions for PostgreSQL:
- [PostgreSQL Aggregate Function Support, page 631](#)
 - [PostgreSQL Analytic Aggregate Function Support, page 631](#)
 - [PostgreSQL Binary Function Support, page 632](#)
 - [PostgreSQL Character Function Support, page 632](#)
 - [PostgreSQL Conversion Function Support, page 633](#)
 - [PostgreSQL Date Function Support, page 634](#)
 - [PostgreSQL Numeric Function Support, page 634](#)
 - [PostgreSQL Time Function Support, page 636](#)

PostgreSQL Aggregate Function Support

TDV supports the aggregate functions listed in the table below for PostgreSQL.

| PostgreSQL Aggregate Function | Notes |
|-------------------------------|---|
| AVG | |
| CORR | |
| COUNT | BLOB, CLOB, and DISTINCT not supported. |
| COVAR_POP | |
| COVAR_SAMP | |
| MAX | |
| MIN | |
| SUM | |
| VARIANCE | |

PostgreSQL Analytic Aggregate Function Support

TDV supports the analytic aggregate functions listed in the table below for PostgreSQL.

| PostgreSQL Analytic Aggregate Function | Notes |
|--|-------|
| CUME_DIST | |
| DENSE_RANK | |
| NTILE | |
| PERCENT_RANK | |
| RANK | |
| ROW_NUMBER | |
| STDDEV | |
| STDDEV_POP | |

| PostgreSQL Analytic Aggregate Function | Notes |
|--|-------|
| STDDEV_SAMP | |
| VAR_POP | |
| VAR_SAMP | |
| VARIANCE_POP | |
| VARIANCE_SAMP | |

PostgreSQL Binary Function Support

TDV supports the binary functions listed in the table below for PostgreSQL.

| PostgreSQL Binary Function | Notes |
|------------------------------------|-------|
| INT1AND, INT2AND, INT4AND, INT8AND | |
| INT1NOT, INT2NOT, INT4NOT, INT8NOT | |
| INT1OR, INT2OR, INT4OR, INT8OR | |
| INT1XOR, INT2XOR, INT4XOR, INT8XOR | |

PostgreSQL Character Function Support

TDV supports the character functions listed in the table below for PostgreSQL.

| PostgreSQL Character Function | Notes |
|-------------------------------|-------|
| ASCII | |
| BTRIM | |
| CHR | |
| CONCAT | |
| INITCAP | |
| LENGTH | |
| LOWER | |

| PostgreSQL Character Function | Notes |
|-------------------------------|-------|
| LPAD | |
| LTRIM | |
| REPLACE | |
| RPAD | |
| RTRIM | |
| STRPOS | |
| SUBSTR | |
| SUBSTRING | |
| TRANSLATE | |
| TRIM | |
| UPPER | |

PostgreSQL Conversion Function Support

TDV supports the conversion functions listed in the table below for PostgreSQL.

| PostgreSQL Conversion Function | Notes |
|--------------------------------|--|
| CAST | <p>PostgreSQL does not preserve trailing spaces when casting CHARs to VARCHARs, so results may differ when a federated data source or TDV is set to honor trailing spaces.</p> <p>PostgreSQL data types have the following restrictions for the CAST function:</p> <ul style="list-style-type: none"> Maximum CHAR length is 2000. Maximum VARCHAR length is 4000. Maximum numeric precision (p) is 38. Maximum numeric scale (s) is 38. |
| FORMAT_DATE | |

| PostgreSQL Conversion Function | Notes |
|--------------------------------|-------|
| PARSE_TIMESTAMP | |
| TO_CHAR | |
| TO_DATE | |
| TO_NUMBER | |
| TO_TIMESTAMP | |

PostgreSQL Date Function Support

TDV supports the date functions listed in the table below for PostgreSQL.

| PostgreSQL Date Function | Notes |
|--------------------------|----------------|
| CLOCK_TIMESTAMP | |
| CURRENT_DATE | |
| CURRENT_TIMESTAMP | |
| DATE_TRUNC | |
| DAY | |
| MONTH | |
| YEAR | Not supported. |

PostgreSQL Numeric Function Support

TDV supports the numeric functions listed in the table below for PostgreSQL.

| PostgreSQL Numeric Function | Notes |
|-----------------------------|-------|
| ABS | |
| ACOS | |
| ASIN | |
| ATAN | |

| PostgreSQL Numeric Function | Notes |
|-----------------------------|---|
| ATAN2 | |
| CEIL | |
| CEILING | |
| COS | |
| COT | |
| DEGREES | |
| EXP | |
| EXTRACT | From a date. |
| FLOOR | |
| LOG | |
| MOD | |
| PI | |
| POW | Not supported: mixing string, number, or NULL with BIT, NCHAR, or NVARCHAR. |
| POWER | |
| RADIANS | |
| RANDOM | |
| ROUND | |
| SIGN | |
| SIN | |
| SQRT | |
| TAN | |

| PostgreSQL Numeric Function | Notes |
|-----------------------------|-------|
| TRUNC | |

PostgreSQL Time Function Support

TDV supports the time functions listed in the table below for PostgreSQL.

| PostgreSQL Time Function | Notes |
|--------------------------|---------------------------------|
| EXTRACT | From TIMESTAMP or INTERVAL_DAY. |
| NOW | |
| TIMEOFDAY | |

Redshift Function Support

TDV supports the following types of functions for Redshift:

- [Redshift Aggregate Function Support, page 636](#)
- [Redshift Analytical Function Support, page 637](#)
- [Redshift Character Function Support, page 637](#)
- [Redshift Conditional Function Support, page 639](#)
- [Redshift Conversion Function Support, page 639](#)
- [Redshift Date Function Support, page 640](#)
- [Redshift Numerical Function Support, page 640](#)
- [Redshift Time Function Support, page 641](#)

Redshift Aggregate Function Support

TDV supports the Aggregate functions listed in the table below for Redshift.

| Redshift Aggregate Function | Notes |
|-----------------------------|---|
| AVG | DISTINCT and ALL supported. |
| COUNT | DISTINCT and ALL supported. BLOB not supported. |

| Redshift Aggregate Function | Notes |
|-----------------------------|---|
| MAX | DISTINCT and ALL supported. NULLs, strings, numbers, dates supported. |
| MIN | DISTINCT supported. |
| SUM | DISTINCT and ALL supported. |

Redshift Analytical Function Support

TDV supports the analytical functions listed in the table below for Redshift.

| Redshift Analytical Function | Notes |
|------------------------------|---|
| AVG | DISTINCT and ALL supported. |
| COUNT | DISTINCT and ALL supported. BLOB not supported. |
| FIRST_VALUE | IGNORENULLS and RESPECTNULLS supported. |
| LAG | IGNORENULLS and RESPECTNULLS supported. |
| LAST_VALUE | IGNORENULLS and RESPECTNULLS supported. |
| LEAD | IGNORENULLS and RESPECTNULLS supported. |
| MAX | DISTINCT and ALL supported. |
| MIN | DISTINCT supported. |
| SUM | DISTINCT and ALL supported. |
| VARIANCE | DISTINCT and ALL supported. Maps to VAR_SAMP. |

Redshift Character Function Support

TDV supports the character functions listed in the table below for Redshift.

| Redshift Character Function | Notes |
|-----------------------------|-------|
| ASCII | |
| BTRIM | |

| Redshift Character Function | Notes |
|-----------------------------|-------|
| CHR | |
| CONCAT | |
| HASHSHA1 | |
| INITCAP | |
| LEAD | |
| LENGTH | |
| LOWER | |
| LPAD | |
| LTRIM | |
| OCTET_LENGTH | |
| POSITION | |
| REPEAT | |
| REPLACE | |
| RPAD | |
| RTRIM | |
| SPACE | |
| STRPOS | |
| SUBSTR | |
| SUBSTRING | |
| TRANSLATE | |
| TRIM | |
| UPPER | |

Redshift Conditional Function Support

TDV supports the conditional functions listed in the table below for Redshift.

| Redshift Conditional Function | Notes |
|-------------------------------|-------|
| CASE | |
| COALESCE | |
| DECODE | |
| NULLIF | |
| NVL | |
| NVL2 | |

Redshift Conversion Function Support

TDV supports the conversion functions listed in the table below for Redshift.

| Redshift Conversion Function | Notes |
|------------------------------|--|
| CAST | <ul style="list-style-type: none">Maximum CHAR length is 10485760Maximum VARCHAR length is 10485760Maximum precision is 38Maximum scale is 38 |
| FORMAT_DATE | |
| PARSE_TIMESTAMP | |
| TO_CHAR | |
| TO_DATE | |
| TO_NUMBER | |
| TO_TIMESTAMP | |

Redshift Date Function Support

TDV supports the date functions listed in the table below for Redshift.

| Redshift Date Function | Notes |
|------------------------|--------------------|
| ADD MONTHS | Maps to ADD_MONTHS |
| CURRENT_DATE | |
| CURRENT_TIMESTAMP | |
| CLOCK_TIMESTAMP | |
| DATEADD | |
| DATEDIFF | |
| DATE_PART | |
| DATE_TRUNC | |
| DAY | |
| MONTH | |
| MONTHS_BETWEEN | |
| TZCONVERTOR | |
| UPDATEDIFF | Maps to DATEDIFF |
| YEAR | |

Redshift Numerical Function Support

TDV supports the numerical functions listed in the table below for Redshift.

| Redshift Numerical Function | Notes |
|-----------------------------|-------|
| ABS | |
| ATAN | |
| ATAN2 | |
| CEIL | |

| Redshift Numerical Function | Notes |
|-----------------------------|-------|
| CEILING | |
| DEGREES | |
| EXP | |
| FACTORIAL | |
| FLOOR | |
| LOG | |
| MOD | |
| PI | |
| POW | |
| POWER | |
| RADIANS | |
| RANDOM | |
| ROUND | |
| SIGN | |
| SIN | |
| SQRT | |
| TAN | |
| TRUNC | |

Redshift Time Function Support

TDV supports the time functions listed in the table below for Redshift.

| Redshift Time Function | Notes |
|------------------------|-------|
| CURRENT_TIME | |
| EXTRACT | |

| Redshift Time Function | Notes |
|------------------------|-------|
| NOW | |
| TIMEOFDAY | |

SAP HANA Function Support

The tables in this section point out where TDV functions (which would appear in views defined in Studio) is mapped to native SAP HANA SPS 09 functions of a different name. For example, the HANA conversion function HEXTOBIN is listed in the Notes column as “Mapped from HEX_TO_BINARY.”

TDV supports the following types of functions for SAP HANA:

- [SAP HANA Aggregate Function Support, page 642](#)
- [SAP HANA Analytical Function Support, page 643](#)
- [SAP HANA Binary Function Support, page 644](#)
- [SAP HANA Character Function Support, page 645](#)
- [SAP HANA Conditional Function Support, page 646](#)
- [SAP HANA Conversion Function Support, page 646](#)
- [SAP HANA Date Function Support, page 647](#)
- [SAP HANA Numeric Function Support, page 648](#)

SAP HANA Aggregate Function Support

TDV supports the aggregate functions listed in the table below for SAP HANA.

| SAP HANA Aggregate Function | Notes |
|-----------------------------|-------|
| AVG | |
| CORR | |
| CORR_SPEARMAN | |
| COUNT | |
| MAX | |

| SAP HANA Aggregate Function | Notes |
|-----------------------------|-------|
| MEDIAN | |
| MIN | |
| STDDEV | |
| SUM | |
| VARIANCE | |

SAP HANA Analytical Function Support

TDV supports the analytical and analytical aggregate functions listed in the table below for SAP HANA.

| SAP HANA Analytical Function | Notes |
|------------------------------|-------|
| AVG | |
| CORR | |
| CORR_SPEARMAN | |
| COUNT | |
| CUME_DIST | |
| DENSE_RANK | |
| FIRST_VALUE | |
| LAG | |
| LAST_VALUE | |
| LEAD | |
| MAX | |
| MEDIAN | |
| MIN | |
| NTILE | |

| SAP HANA Analytical Function | Notes |
|------------------------------|-----------------------|
| PERCENT_RANK | |
| PERCENTILE_CONT | |
| PERCENTILE_DISC | |
| RANK | |
| ROW_NUMBER | |
| STDDEV | |
| SUM | |
| VAR | Mapped from VARIANCE. |

SAP HANA Binary Function Support

TDV supports the binary functions listed in the table below for SAP HANA.

| SAP HANA Binary Function | Notes |
|--------------------------|---|
| BITAND | Mapped from INT1AND, INT2AND, INT4AND, INT8AND. |
| BITCOUNT | |
| BITNOT | Mapped from INT1NOT, INT2NOT, INT4NOT, INT8NOT. |
| BITOR | Mapped from INT1OR, INT2OR, INT4OR, INT8OR. |
| BITXOR | Mapped from INT1XOR, INT2XOR, INT4XOR, INT8XOR. |

SAP HANA Character Function Support

TDV supports the character functions listed in the table below for SAP HANA.

| SAP HANA Character Function | Notes |
|-----------------------------|---|
| ASCII | |
| CHAR | Mapped from CHR. |
| CONCAT | |
| LCASE | |
| LEFT | |
| LENGTH | Mapped from CHAR_LENGTH, CHARACTER_LENGTH, or LENGTH. |
| LIKE_REGEX | Mapped from REGEXP. |
| LOCATE | Mapped from FIND, INSTR, LOCATE, or POSITION. |
| LOWER | |
| LPAD | |
| LTRIM | |
| NCHAR | Mapped from UNICHR. |
| REPLACE | |
| REPLACE_REGEXPR | Mapped from REGEXP_REPLACE. |
| RIGHT | |
| RPAD | |
| RTRIM | |
| SUBSTR | |
| SUBSTR_REGEXPR | Mapped from REGEXP_EXTRACT. |

| SAP HANA Character Function | Notes |
|-----------------------------|--|
| SUBSTRING | In the 3-argument form, if the second argument is 0 or negative, SAP HANA's results deviate from SQL standard. To prevent nonstandard results, add OPTION STRICT to the query. |
| TRIM | |
| UCASE | |
| UNICODE | |
| UPPER | |

SAP HANA Conditional Function Support

TDV supports the conditional functions listed in the table below for SAP HANA.

| SAP HANA Conditional Function | Notes |
|-------------------------------|-------|
| COALESCE | |
| GREATEST | |
| IFNULL | |
| LEAST | |
| NULLIF | |

SAP HANA Conversion Function Support

TDV supports the conversion functions listed in the table below for SAP HANA.

| SAP HANA Conversion Function | Notes |
|------------------------------|----------------------------|
| BINTOHEX | Mapped from TO_HEX. |
| CAST | |
| HEXTOBIN | Mapped from HEX_TO_BINARY. |

| SAP HANA Conversion Function | Notes |
|------------------------------|--|
| TO_DATE | |
| TO_TIMESTAMP | |
| TO_VARCHAR | Mapped from FORMAT_DATE, PARSE_DATE, PARSE_TIME PARSE_TIMESTAMP, or TO_CHAR. |

SAP HANA Date Function Support

TDV supports the date functions listed in the table below for SAP HANA.

| SAP HANA Date Function | Notes |
|------------------------|--------------------------------|
| ADD_DAYS | Mapped from DATE_ADD. |
| ADD_MONTH | |
| CURRENT_DATE | |
| CURRENT_TIME | |
| CURRENT_TIMESTAMP | |
| CURRENT_UTCTIMESTAMP | Mapped from GETUTCDATE. |
| DAYNAME | |
| DAYOFMONTH | |
| DAYOFYEAR | Mapped from EXTRACT (DOYFROM). |
| DAYS_BETWEEN | |
| EXTRACT (DAY FROM) | |
| EXTRACT (HOUR FROM) | |
| EXTRACT (MINUTE FROM) | |
| EXTRACT (MONTH FROM) | |
| EXTRACT (SECOND FROM) | |
| EXTRACT (YEAR FROM) | |

| SAP HANA Date Function | Notes |
|------------------------|------------------------|
| HOUR | |
| LAST_DAY | |
| MINUTE | |
| MONTH | |
| MONTHNAME | |
| NOW | |
| QUARTER | |
| SECOND | |
| WEEK | |
| WEEKDAY | Mapped from DAYOFWEEK. |
| YEAR | |

SAP HANA Numeric Function Support

TDV supports the numeric functions listed in the table below for SAP HANA.

| SAP HANA Numeric Function | Notes |
|---------------------------|------------------------------|
| ABS | |
| ACOS | |
| ASIN | |
| ATAN | |
| ATAN2 | |
| CEIL | Mapped from CEIL or CEILING. |
| COS | |
| COT | |
| EXP | |

| SAP HANA Numeric Function | Notes |
|---------------------------|-----------------------------|
| FLOOR | |
| LN | |
| LOG | |
| MOD | |
| POWER | Mapped from POW or POWER. |
| RAND | Mapped from RAND or RANDOM. |
| ROUND | |
| SIGN | |
| SIN | |
| SQRT | |
| TAN | |

Sybase Function Support

TDV supports the following types of functions for Sybase:

- [Sybase Aggregate Function Support, page 650](#)
- [Sybase Character Function Support, page 650](#)
- [Sybase Conditional Function Support, page 651](#)
- [Sybase Conversion Function Support, page 651](#)
- [Sybase Date Function Support, page 651](#)
- [Sybase Numeric Function Support, page 652](#)

A further section describes a workaround:

- [Sybase ASE 15.7 MERGE Behavior, page 653](#)

Sybase Aggregate Function Support

TDV supports the aggregate functions listed in the table below for Sybase.

| Sybase Aggregate Function | Notes |
|---------------------------|--|
| AVG | String-type arguments not supported; DISTINCT supported. |
| COUNT | DISTINCT supported. |
| MAX | DISTINCT supported. |
| MIN | DISTINCT supported. |
| PERCENTILE_CONT | |
| PERCENTILE_DISC | |
| SUM | String-type arguments not supported; DISTINCT supported. |

Sybase Character Function Support

TDV supports the character functions listed in the table below for Sybase.

| Sybase Character Function | Notes |
|---------------------------|-------|
| CONCAT | |
| LENGTH | |
| LOWER | |
| POSITION | |
| RTRIM | |
| SUBSTRING | |
| TRIM | |
| UPPER | |

Sybase Conditional Function Support

TDV supports the conditional functions listed in the table below for Sybase.

| Sybase Conditional Function | Notes |
|-----------------------------|--|
| COALESCE | |
| DECODE | Mapped to CASE. |
| DECODE | Mapped to CASE. |
| NULLIF | NULL literal cannot be the first argument; IMAGE, NTEXT, TEXT not allowed. |

Sybase Conversion Function Support

TDV supports the conversion functions listed in the table below for Sybase.

| Sybase Conversion Function | Notes |
|----------------------------|----------------|
| CAST | |
| FORMAT_DATE | |
| PARSE_DATE | Does not push. |
| PARSE_TIME | Does not push. |
| PARSE_TIMESTAMP | |
| TO_CHAR | |
| TO_DATE | |
| TO_NUMBER | |
| TO_TIMESTAMP | |

Sybase Date Function Support

TDV supports the date functions listed in the table below for Sybase.

| Sybase Date Function | Notes |
|----------------------|-------|
| CURRENT_TIMESTAMP | |

| Sybase Date Function | Notes |
|----------------------|---|
| DATEDIFF | DATEDIFF (datepart, startdate, enddate) Sybase produces correct (standard) results for year, month, day date parts and <i>incorrect</i> results for hour, minute, second date parts. |
| DAY | |
| MONTH | |
| YEAR | |

Sybase Numeric Function Support

TDV supports the numeric functions listed in the table below for Sybase. Sybase does not allow string arguments for these functions.

| Sybase Numeric Function | Notes |
|-------------------------|----------------|
| ABS | |
| ACOS | |
| ASIN | |
| ATAN | |
| CEILING | |
| COS | |
| COT | |
| DEGREES | |
| EXP | |
| FLOOR | |
| LOG | |
| PI | Not supported. |
| POWER | |

| Sybase Numeric Function | Notes |
|-------------------------|-------|
| RADIANS | |
| ROUND | |
| SIN | |
| SQRT | |
| TAN | |

Sybase ASE 15.7 MERGE Behavior

If you have a MERGE statement with *only* a DELETE action, Sybase ASE 15.7 throws the following exception:

```
Caused by: com.sybase.jdbc3.jdbc.SybSQLException: Incomplete MERGE statement
SQL State = ZZZZZ SQL Error Code = 3640
    at com.sybase.jdbc3.tds.Tds.a(Unknown Source)
```

The workaround is to add a second, “placeholder” action.

For example, the following fails:

```
MERGE INTO /users/composite/test/sources/mergeSyntax/sybase15/mergedb/msbe002/STATION_COPY USING
/users/composite/test/sources/mergeSyntax/sybase15/mergedb/msbe002/STATION ON
(STATION_COPY.ID = STATION.ID) AND STATION.CITY='Denver' WHEN MATCHED THEN DELETE
```

But the following succeeds:

```
MERGE INTO /users/composite/test/sources/mergeSyntax/sybase15/mergedb/msbe002/STATION_COPY USING
/users/composite/test/sources/mergeSyntax/sybase15/mergedb/msbe002/STATION ON (STATION_COPY.ID =
STATION.ID) AND STATION.CITY='Denver'
WHEN MATCHED THEN DELETE
WHEN NOT MATCHED AND 1<>1 THEN INSERT (ID, MONTHS) VALUES (1, 2)
```

Sybase IQ Function Support

TDV supports the following types of functions for Sybase IQ:

- [Sybase IQ Aggregate Function Support, page 654](#)
- [Sybase IQ Analytic Function Support, page 655](#)
- [Sybase IQ Character Function Support, page 655](#)
- [Sybase IQ Conditional Function Support, page 656](#)
- [Sybase IQ Conversion Function Support, page 656](#)

- [Sybase IQ Date Function Support, page 657](#)
- [Sybase IQ Numeric Function Support, page 657](#)

Sybase IQ Aggregate Function Support

TDV supports the aggregate functions listed in the table below for Sybase IQ.

| Sybase IQ Aggregate Function | Notes |
|------------------------------|---|
| AVG | String-type arguments not supported. |
| CORR | |
| COUNT | DISTINCT not supported. |
| COVAR_POP | |
| COVAR_SAMP | |
| MAX | DISTINCT supported. |
| MEDIAN | |
| MIN | DISTINCT supported. |
| STDDEV | String, or number + DISTINCT, or string + DISTINCT not supported. |
| STDDEV_POP | Number + DISTINCT not supported. |
| STDDEV_SAMP | |
| SUM | String-type arguments not supported. DISTINCT supported. |
| VAR_POP | Number + DISTINCT not supported. |
| VAR_SAMP | |
| VARIANCE | String, or number, or string + DISTINCT not supported. |

Sybase IQ Analytic Function Support

TDV supports the analytic functions listed in the table below for Sybase IQ.

| Sybase IQ Analytic Function | Notes |
|-----------------------------|--|
| AVG | Whole number + DISTINCT, and floating point + DISTINCT: not supported. |
| DENSE_RANK | |
| EXP_WEIGHTED_AVG | |
| FIRST_VALUE | |
| LAG | |
| LAST_VALUE | |
| LEAD | |
| NTILE | |
| PERCENT_RANK | |
| PERCENTILE_CONT | |
| PERCENTILE_DISC | |
| RANK | |
| ROW_NUMBER | |

Sybase IQ Character Function Support

TDV supports the character functions listed in the table below for Sybase IQ.

| Sybase IQ Character Function | Notes |
|------------------------------|-------|
| CONCAT | |
| LENGTH | |
| LOWER | |
| POSITION | |

| Sybase IQ Character Function | Notes |
|------------------------------|---|
| RTRIM | |
| SUBSTRING | If the value of the third argument (length) is greater than 33000, Sybase IQ returns NULL. (TDV server correctly returns the value to the end of the string.) |
| TRIM | |
| UPPER | |

Sybase IQ Conditional Function Support

TDV supports the conditional functions listed in the table below for Sybase IQ.

| Sybase IQ Conditional Function | Notes |
|--------------------------------|--|
| COALESCE | |
| DECODE | Mapped to CASE. |
| NULLIF | NULL literal cannot be the first argument; IMAGE, NTEXT, TEXT not allowed. The following first-second argument combinations are not supported: STRING-LONGVARCHAR, LONGVARCHAR-STRING, LONGVARCHAR-DATE, LONGVARCHAR-NULL, DATE-LONGVARCHAR. |

Sybase IQ Conversion Function Support

TDV supports the conversion functions listed in the table below for Sybase IQ.

| Sybase IQ Conversion Function | Notes |
|-------------------------------|-------|
| CAST | |
| FORMAT_DATE | |
| PARSE_TIMESTAMP | |
| TO_CHAR | |
| TO_DATE | |

| Sybase IQ Conversion Function | Notes |
|-------------------------------|-------|
| TO_NUMBER | |
| TO_TIMESTAMP | |

Sybase IQ Date Function Support

TDV supports the date functions listed in the table below for Sybase IQ.

| Sybase IQ Date Function | Notes |
|-------------------------|--|
| CURRENT_TIMESTAMP | |
| DATEDIFF | DATEDIFF (datepart, startdate, enddate) Sybase IQ produces correct (standard) results for year, month, day date parts and <i>incorrect</i> results for hour, minute, and second date parts. |
| DAY | |
| MONTH | |
| YEAR | |

Sybase IQ Numeric Function Support

TDV supports the numeric functions listed in the table below for Sybase IQ.
Sybase IQ does not allow string arguments for numeric functions.

| Sybase IQ Numeric Function | Notes |
|----------------------------|-------|
| ABS | |
| ACOS | |
| ASIN | |
| ATAN | |
| CEILING | |
| COS | |
| COT | |

| Sybase IQ Numeric Function | Notes |
|----------------------------|---|
| DEGREES | |
| EXP | |
| FLOOR | |
| LOG | |
| PI | Not supported. |
| POWER | Returns NULL instead of infinity if second argument exceeds the maximum floating point value. |
| RADIANS | |
| ROUND | |
| SIN | |
| SQRT | |
| TAN | |

Teradata Function Support

TDV supports the following types of functions for Teradata:

- [Teradata Aggregate Function Support, page 659](#)
- [Teradata Analytic Function Support, page 659](#)
- [Teradata Character Function Support, page 660](#)
- [Teradata Conditional Function Support, page 661](#)
- [Teradata Conversion Function Support, page 662](#)
- [Teradata Date Function Support, page 663](#)
- [Teradata Number Function Support, page 663](#)

Teradata Aggregate Function Support

TDV supports the aggregate functions listed in the table below for Teradata.

| Teradata Aggregate Function | Notes |
|-----------------------------|---|
| AVG | |
| COUNT | Large objects (BLOB, CLOB, and so on) not supported. |
| MAX | Decimal not supported (may introduce rounding error). |
| MEDIAN | Version 15 only. |
| MIN | Decimal not supported (may introduce rounding error). |
| SUM | |

Teradata Analytic Function Support

TDV supports the analytic functions listed in the table below for Teradata.

| Teradata Analytic Function | Notes |
|----------------------------|---|
| AVG | |
| COUNT | Large objects (BLOB, CLOB, and so on) not supported. Version 15: JSON not supported. |
| CUME_DIST | Version 15 only. |
| DENSE_RANK | Version 15 only. |
| FIRST_VALUE | Includes ignore nulls. Version 15 only. |
| LAST_VALUE | Includes ignore nulls. Version 15 only. |
| MAX | Decimal not supported (may introduce rounding error). Version 15: JSON not supported. |

| Teradata Analytic Function | Notes |
|----------------------------|---|
| MIN | Decimal not supported (may introduce rounding error). MIN (date) incorrectly returns NULL if one of the date columns is NULL. Version 15: JSON not supported. |
| PERCENT_RANK | |
| PERCENTILE_CONT | Version 15 only. |
| PERCENTILE_DIST | Version 15 only. |
| RANK | |
| ROW_NUMBER | |
| STDDEV_POP | |
| STDDEV_SAMP | |
| SUM | |
| VAR_POP | |
| VAR_SAMP | |
| VARIANCE_POP | |
| VARIANCE_SAMP | |

Teradata Character Function Support

TDV supports the character functions listed in the table below for Teradata.

| Teradata Character Function | Notes |
|-----------------------------|------------------|
| ASCII | Version 15 only. |
| CHAR_LENGTH | |
| CHR | Version 15 only. |
| CONCAT | |
| INITCAP | Version 15 only. |

| Teradata Character Function | Notes |
|-----------------------------|------------------|
| INSTR | Version 15 only. |
| LEAD | Version 15 only. |
| LEFT | Version 15 only. |
| LENGTH | |
| LOWER | |
| LPAD | Version 15 only. |
| LTRIM | Version 15 only. |
| POSITION | |
| REPLACE | Version 15 only. |
| REVERSE | Version 15 only. |
| RIGHT | Version 15 only. |
| RPAD | Version 15 only. |
| RTRIM | |
| SPACE | Not supported. |
| SUBSTRING | |
| TRANSLATE | Version 15 only. |
| TRIM | |
| UPPER | |

Teradata Conditional Function Support

TDV supports the conditional functions listed in the table below for Teradata.

| Teradata Conditional Function | Notes |
|-------------------------------|------------------|
| COALESCE | |
| DECODE | Version 15 only. |

| Teradata Conditional Function | Notes |
|-------------------------------|------------------|
| GREATEST | Version 15 only. |
| LEAST | Version 15 only. |
| NULLIF | |
| NVL | |
| NVL2 | Version 15 only. |

Teradata Conversion Function Support

TDV supports the conversion functions listed in the table below for Teradata.

| Teradata Conversion Function | Notes |
|------------------------------|---|
| CAST | Not supported in versions lower than v12/v13: STRING or NUMBER is cast as BIGINT. |
| FORMAT | |
| FORMAT_DATE | |
| PARSE_DATE | |
| PARSE_TIME | |
| PARSE_TIMESTAMP | |
| TO_CHAR | Version 15 only. Output format may differ for push versus no-push, but precision and scale are the same. For example, Teradata 15 TDV might return 1.23456789000000E 005, while TDV (DISABLE_PUSH='TRUE') returns 123456.789. |
| TO_DATE | |
| TO_TIMESTAMP | Version 15 only. |
| TO_TIMESTAMP_TZ | Version 15 only. |

Teradata Date Function Support

TDV supports the date functions listed in the table below for Teradata.

| Teradata Date Function | Notes |
|------------------------|--|
| CURRENT_DATE | |
| CURRENT_TIME | |
| CURRENT_TIMESTAMP | |
| DAY | |
| EXTRACTDAY | |
| EXTRACTHOUR | Not supported: EXTRACTHOUR from INTERVAL_DAY or INTERVAL_MINUTE. |
| EXTRACTMINUTE | |
| EXTRACTMONTH | |
| EXTRACTSECOND | |
| EXTRACTYEAR | |
| LAST_DAY | Version 15 only. |
| MONTH | |
| MONTHS_BETWEEN | Version 15 only. |
| NEXT_DAY | Version 15 only. |
| YEAR | |

Teradata Number Function Support

TDV supports the number functions listed in the table below for Teradata, except as marked.

| Teradata Number Function | Notes |
|--------------------------|-------|
| ABS | |
| ACOS | |

| Teradata Number Function | Notes |
|--------------------------|--------------------------|
| ASIN | |
| ATAN | |
| CEILING | Versions 13 and 15 only. |
| COS | |
| COT | |
| DEGREE | Not supported. |
| EXP | |
| FLOOR | Versions 13 and 15 only. |
| LOG | |
| PI | Not supported. |
| RADIANS | Not supported. |
| ROUND | Version 15 only. |
| SIGN | Version 15 only. |
| SIN | |
| SQRT | |
| TAN | |
| TRUNC | Version 15 only. |

Vertica Function Support

TDV supports the following types of functions for Vertica:

- [Vertica Aggregate Function Support, page 665](#)
- [Vertica Analytic Function Support, page 667](#)
- [Vertica Binary Function Support, page 668](#)
- [Vertica Character Function Support, page 668](#)

- [Vertica Conditional Function Support, page 671](#)
- [Vertica Conversion Function Support, page 671](#)
- [Vertica Date Function Support, page 672](#)
- [Vertica Numeric Function Support, page 675](#)
- [Vertica OLAP Analytic Function Support, page 676](#)
- [Vertica Time Series Function Support, page 678.](#)

All aggregate, date/time, formatting, and string functions are supported in pass-through mode.

In Vertica 6.1, TDV supports:

- EXCEPT operators
- INTERSECT operators
- WITH clause, with or without column aliasing

In Vertica 5.0 or 6.1, TDV supports:

- Queries with a WHERE filter on date, time, and timestamp columns
- Subqueries in EXISTS clauses
- Subqueries in IN clauses

TDV also supports the LIMIT clause in Vertica, but maps it to TDV syntax. For example:

```
SELECT * FROM tableA LIMIT 3 OFFSET 2
```

This is implemented with the syntax:

```
SELECT * FROM tableA OFFSET 2 ROWS FETCH 3 ROWS ONLY
```

Vertica Aggregate Function Support

TDV supports the aggregate functions listed in the table below for Vertica.

| Vertica Aggregate Function | Notes |
|----------------------------|-------|
| AVG | |
| CORR | |
| COUNT | |
| COVAR_POP | |

| Vertica Aggregate Function | Notes |
|----------------------------|-------|
| COVAR_SAMP | |
| MAX | |
| MIN | |
| PERCENTILE_CONT | |
| PERCENTILE_DISC | |
| REGR_AVGX | |
| REGR_AVGY | |
| REGR_COUNT | |
| REGR_INTERCEPT | |
| REGR_R2 | |
| REGR_SLOPE | |
| REGR_SXX | |
| REGR_SXY | |
| REGR_SYY | |
| STDDEV | |
| STDDEV_POP | |
| STDDEV_SAMP | |
| SUM | |
| SUM_FLOAT | |
| VAR_POP | |
| VAR_SAMP | |
| VARIANCE | |
| VARIANCE_POP | |

| Vertica Aggregate Function | Notes |
|----------------------------|-------|
| VARIANCE_SAMP | |

Vertica Analytic Function Support

TDV supports the analytic functions listed in the table below for Vertica.

| Vertica Analytic Function | Notes |
|----------------------------|-------|
| AVG | |
| CORR | |
| COUNT | |
| COVAR_POP | |
| COVAR_SAMP | |
| CUME_DIST | |
| DENSE_RANK | |
| EXPONENTIAL_MOVING_AVERAGE | . |
| FIRST_VALUE | |
| LAG | |
| LAST_VALUE | |
| LEAD | |
| MAX | |
| MEDIAN | |
| MIN | |
| NTILE | |
| PERCENT_RANK | |
| PERCENTILE_CONT | |
| PERCENTILE_DISC | |

| Vertica Analytic Function | Notes |
|---------------------------|-------|
| RANK | |
| ROW_NUMBER | |
| STDDEV | |
| STDDEV_POP | |
| STDDEV_SAMP | |
| SUM | |
| VAR_POP | |
| VAR_SAMP | |
| VARIANCE | |

Vertica Binary Function Support

TDV supports the binary functions listed in the table below for Vertica.

| Vertica Binary Function | Notes |
|------------------------------------|-------|
| INT1AND, INT2AND, INT4AND, INT8AND | |
| INT1NOT, INT2NOT, INT4NOT, INT8NOT | |
| INT1OR, INT2OR, INT4OR, INT8OR | |
| INT1SHL, INT2SHL, INT4SHL, INT8SHL | |
| INT1SHR, INT2SHR, INT4SHR, INT8SHR | |
| INT1XOR, INT2XOR, INT4XOR, INT8XOR | |

Vertica Character Function Support

TDV supports the character functions listed in the table below for Vertica.

| Vertica Character Function | Notes |
|----------------------------|-------|
| ASCII | |

| Vertica Character Function | Notes |
|----------------------------|-------------------------------|
| BITCOUNT | |
| BITSTRING_TO_BINARY | |
| BIT_AND | |
| BIT_LENGTH | |
| BIT_OR | |
| BIT_XOR | |
| BTRIM | |
| CHAR_LENGTH | |
| CHARACTER_LENGTH | |
| CHR | |
| CONCAT | |
| GREATEST | |
| NEX_TO_BINARY | |
| INET_ATON | |
| INET_NTOA | |
| INITCAP | |
| INSERT | String-manipulation function. |
| INSTR | |
| ISUTF8 | |
| LEAST | |
| LEFT | |
| LENGTH | |
| LOWER | |

| Vertica Character Function | Notes |
|----------------------------|-------|
| LPAD | |
| LTRIM | |
| MD5 | |
| OCTET_LENGTH | |
| OVERLAYB | |
| POSITION | |
| QUOTE_IDENT | |
| QUOTE_LITERAL | |
| REGEXP_REPLACE | |
| REPEAT | |
| REPLACE | |
| RIGHT | |
| RPAD | |
| RTRIM | |
| SPACE | |
| SPLIT_PART | |
| STRPOS | |
| SUBSTR | |
| SUBSTRING | |
| TRANSLATE | |
| TRIM | |
| TRUNC | |
| UPPER | |

| Vertica Character Function | Notes |
|----------------------------|-------|
| V6_ATON | |
| V6_NTOA | |
| V6_SUBNETA | |
| V6_SUBNETN | |
| V6_TYPE | |

Vertica Conditional Function Support

TDV supports the conditional functions listed in the table below for Vertica.

| Vertica Conditional Function | Notes |
|------------------------------|-------|
| COALESCE | |
| CONDITIONAL_CHANGE_EVENT | |
| CONDITIONAL_TRUE_EVENT | |
| DECODE | |
| IFNULL | |
| ISNULL | |
| NULLIF | |
| NVL | |
| NVL2 | |

Vertica Conversion Function Support

TDV supports the conversion functions listed in the table below for Vertica.

| Vertica Conversion Function | Notes |
|-----------------------------|-------|
| CAST | |
| TO_BITSTRING | |

| Vertica Conversion Function | Notes |
|-----------------------------|-------------------|
| TO_CHAR | |
| TO_DATE | |
| TO_HEX | |
| TO_NUMBER | |
| TO_TIMESTAMP | |
| TO_TIMESTAMP_TZ | Vertica 6.1 only. |

Vertica Date Function Support

- TDV supports the date functions listed in the table below for Vertica.
- With Vertica date functions, push results may differ from no-push results. For example:
- Although Vertica accepts `TIMESTAMP` as an argument for `MONTHS_BETWEEN`, it ignores the time part while calculating. TDV honors the time part while calculating.
 - `DATEDIFF` calculates results according to ticks (boundaries) crossed within a date or time range (counting the enddate but not the startdate). For years, the boundary is January 1. Months are based on calendar months, not the days within months. Weeks start at midnight on Sunday. Days are based on calendar days, not the hours within days, and so on.

| Vertica Date Function | Notes |
|-----------------------|-------|
| ADD_MONTHS | |
| AT TIME ZONE | |
| CLOCK_TIMESTAMP | |
| CURRENT_DATE | |
| CURRENT_TIME | |
| CURRENT_TIMESTAMP | |
| DATEDIFF | |

| Vertica Date Function | Notes |
|-----------------------|---|
| DATE_PART | |
| DATE_TRUNC | |
| DAY | |
| DAYOFMONTH | |
| DAYOFWEEK | |
| DAYOFWEEK_ISO | |
| DAYOFYEAR | |
| DAYS | |
| EXTRACT | . |
| GETDATE | Not supported. Use SYSDATE (identical). |
| GETUTCDATE | |
| HOUR | |
| ISFINITE | |
| JULIAN_DAY | |
| LAST_DAY | |
| LOCALTIME | |
| LOCALTIMESTAMP | |
| MICROSECOND | |
| MIDNIGHT_SECONDS | |
| MINUTE | |
| MONTH | |
| MONTHS_BETWEEN | Because |
| NEW_TIME | Vertica 6.1 only. |

| Vertica Date Function | Notes |
|-----------------------|--|
| NEXT_DAY | |
| NOW | |
| QUARTER | |
| ROUND | For date/time arguments. |
| SECOND | |
| STATEMENT_TIMESTAMP | |
| SYSDATE | |
| TIMEOFDAY | |
| TIMESTAMPADD | <div>Keywords that TDV requires in the TDV source code as the first argument (no quotation marks):<ul style="list-style-type: none">SQL_TSI_FRAC_SECONDSQL_TSI_SECONDSQL_TSI_MINUTESQL_TSI_HOURSQL_TSI_DAYSQL_TSI_WEEKSQL_TSI_MONTHSQL_TSI_QUARTERSQL_TSI_YEAR</div> |
| TIMESTAMP_ROUND | |
| TIMESTAMP_TRUNC | |

| Vertica Date Function | Notes |
|-----------------------|--|
| TIMESTAMPDIFF | Keywords that TDV requires in the TDV source code as the first argument (no quotation marks): <ul style="list-style-type: none">SQL_TSI_FRAC_SECONDSQL_TSI_SECONDSQL_TSI_MINUTESQL_TSI_HOURSQL_TSI_DAYSQL_TSI_WEEKSQL_TSI_MONTHSQL_TSI_QUARTERSQL_TSI_YEAR |
| TRUNC | |
| WEEK | |
| WEEK_ISO | |
| YEAR | |
| YEAR_ISO | |

Vertica Numeric Function Support

TDV supports the numeric functions listed in the table below for Vertica.

| Vertica Numeric Function | Notes |
|--------------------------|-------|
| ABS | |
| ACOS | |
| ASIN | |
| ATAN | |
| ATAN2 | |

| Vertica Numeric Function | Notes |
|--------------------------|-------|
| CBRT | |
| CEILING | |
| COS | |
| COT | |
| DEGREES | |
| EXP | |
| FLOOR | |
| LN | |
| LOG | |
| MOD | |
| PI | |
| POWER | |
| RADIANS | |
| RANDOM | |
| ROUND | |
| SIGN | |
| SIN | |
| SQRT | |
| TAN | |

Vertica OLAP Analytic Function Support

TDV supports the Vertica OLAP analytic functions shown in the table below. These analytic functions are supported in pass-through mode.

Each of these functions returns the same number of rows as the input. These functions operate on groups of rows defined by frame clauses and window partitioning rather than by a GROUP BY clause.

| Vertica OLAP Analytic Function | Notes |
|--------------------------------|-------|
| AVG | |
| COUNT | |
| CUME_DIST | |
| DENSE_RANK | |
| EXPONENTIAL_MOVING_AVERAGE | |
| FIRST_VALUE | |
| LAG | |
| LAST_VALUE | |
| LEAD | |
| MAX | |
| MEDIAN | |
| MIN | |
| NTILE | |
| PERCENT_RANK | |
| PERCENTILE_CONT | |
| PERCENTILE_DISC | |
| RANK | |
| ROW_NUMBER | |
| STDDEV | |
| STDDEV_POP | |
| STDDEV_SAMP | |

| Vertica OLAP Analytic Function | Notes |
|--------------------------------|-------|
| SUM | |
| VAR_POP | |
| VAR_SAMP | |
| VARIANCE | |

Vertica Time Series Function Support

TDV supports the Vertica time series functions listed in the table below, which are used in TIMESERIES clauses. For a complete description of the functions, see the *Vertica Analytic Database SQL Reference Manual*. For details on how to use them with TDV, see “Data Ship Performance Optimization” in the *TDV User Guide*.

| Vertica Time Series Function | Notes |
|------------------------------|-------|
| TIME_SLICE | |
| TIMESERIES | |
| TO_FIRST_VALUE | |
| TO_LAST_VALUE | |
| TS_FIRST_VALUE | |
| TS_LAST_VALUE | |

XML Function Support

TDV supports the following types of functions for XML data sources:

- [XML Aggregate Function Support, page 679](#)
- [XML Character Function Support, page 679](#)
- [XML Conversion Function Support, page 680](#)
- [XML Date Function Support, page 680](#)
- [XML Numeric Function Support, page 680](#)

XML Aggregate Function Support

TDV supports the aggregate functions listed in the table below for XML data sources.

| XML Aggregate Function | Notes |
|------------------------|-------|
| AVG | |
| COUNT | |
| MAX | |
| MIN | |
| SUM | |

XML Character Function Support

TDV supports the character functions listed in the table below for XML data sources.

| XML Character Function | Notes |
|------------------------|-------|
| CONCAT | |
| LENGTH | |
| LOWER | |
| REPLACE | |
| RTRIM | |
| SUBSTRING | |
| TRIM | |
| UPPER | |

XML Conversion Function Support

TDV supports the conversion functions listed in the table below for XML data sources.

| XML Conversion Function | Notes |
|-------------------------|-------|
| CAST | |
| TO_CHAR | |
| TO_DATE | |
| TO_NUMBER | |
| TO_TIMESTAMP | |

XML Date Function Support

TDV supports the date functions listed in the table below for XML data sources.

| XML Date Function | Notes |
|-------------------|-------|
| CURDAY | |
| CURTIME | |
| CURTIMESTAMP | |
| DAY | |
| MONTH | |
| YEAR | |

XML Numeric Function Support

TDV supports the numeric functions listed in the table below for XML data sources.

| XML Numeric Function | Notes |
|----------------------|-------|
| ABS | |
| ACOS | |

| XML Numeric Function | Notes |
|----------------------|-------|
| ASIN | |
| ATAN | |
| CEILING | |
| COS | |
| COT | |
| DEGREES | |
| EXP | |
| FLOOR | |
| LOG | |
| PI | |
| POWER | |
| RADIANS | |
| ROUND | |
| SIN | |
| SQRT | |
| TAN | |

Custom Procedure Examples

This topic contains several examples to illustrate the behavior of a custom procedure. All examples are written in Java for execution on a Windows platform.

- [About the Custom Procedure Examples Syntax, page 683](#)
- [Example 1: Simple Query, page 683](#)
- [Example 2: Simple Update, page 686](#)
- [Example 3: External Update without Compensation, page 689](#)
- [Example 4: Nontransactional External Update without Compensation, page 694](#)
- [Example 5: Expression Evaluator, page 698](#)
- [Example 6: Output Cursor, page 702](#)
- [Example 7: Simple Procedure that Invokes Another Procedure, page 706](#)

About the Custom Procedure Examples Syntax

Developers creating procedures for execution on a UNIX or Linux operating system need to use colons (instead of semicolons) as separators. Also when using new line strings, for Windows it will be “/r/n” compared with Linux “/n”.

Regardless of the operating system, path names must use the forward slash. For example:

```
// Update in the first data source using a SQL statement
numRowsUpdated = qenv.executeUpdate(
    "UPDATE /shared/tutorial/sources/ds_orders/customers" +
    " SET ContactFirstName='" + inputValues[1] +
    "', ContactLastName='" + inputValues[2] +
    "', CompanyName='" + inputValues[3] +
    "', PhoneNumber='" + inputValues[4] +
    "' WHERE CustomerID=" + inputValues[0],
    null);
```

Example 1: Simple Query

This custom procedure participates in the parent transaction, and invokes a query using the execution environment.

```
package proc;
```

```

import com.compositesw.extension.*;
import java.sql.*;

public class SimpleQuery

    implements CustomProcedure

{
    private ExecutionEnvironment qenv;
    private ResultSet resultSet;

    public SimpleQuery() { }

    /**
     * This is called once just after constructing the class. The
     * environment contains methods used to interact with the server.
     */

    public void initialize(ExecutionEnvironment qenv) {
        this.qenv = qenv;
    }

    /**
     * Called during introspection to get the description of the input
     * and output parameters. Should not return null.
     */

    public ParameterInfo[] getParameterInfo() {

        return new ParameterInfo[] {
            new ParameterInfo("id", Types.INTEGER, DIRECTION_IN),
            new ParameterInfo("result", TYPED_CURSOR, DIRECTION_OUT,
                new ParameterInfo[] {
                    new ParameterInfo("Id", Types.INTEGER, DIRECTION_NONE),
                    new ParameterInfo("FirstName", Types.VARCHAR, DIRECTION_NONE),
                    new ParameterInfo("LastName", Types.VARCHAR, DIRECTION_NONE),
                    new ParameterInfo("CompanyName", Types.VARCHAR, DIRECTION_NONE),
                    new ParameterInfo("PhoneNumber", Types.VARCHAR, DIRECTION_NONE),
                }
            )
        };
    }

    /**
     * Called to invoke the stored procedure. Will only be called a
     * single time per instance. Can throw CustomProcedureException or
     * SQLException if there is an error during invoke.
     */

    public void invoke(Object[] inputValues)
        throws CustomProcedureException, SQLException
    {
        resultSet = qenv.executeQuery(
            "SELECT " +
            "CustomerID AS Id, " +
            "ContactFirstName AS FirstName, " +
            "ContactLastName AS LastName, " +

```



```

        "CompanyName AS CompanyName, " +
        "PhoneNumber AS PhoneNumber FROM " +
        "/shared/tutorial/sources/ds_orders/customers WHERE CustomerID=" +
        inputValues[0],
        null);
    }
    /**
     * Called to retrieve the number of rows that were inserted,
     * updated, or deleted during the execution of the procedure. A
     * return value of -1 indicates that the number of affected rows is
     * unknown. Can throw CustomProcedureException or SQLException if
     * there is an error when getting the number of affected rows.
     */
    public int getNumAffectedRows() {
        return 0;
    }

    /**
     * Called to retrieve the output values. The returned objects
     * should obey the Java to SQL typing conventions as defined in the
     * table above. Output cursors can be returned as either
     * CustomCursor or java.sql.ResultSet. Can throw
     * CustomProcedureException or SQLException if there is an error
     * when getting the output values. Should not return null.
     */

    public Object[] getOutputValues() {
        return new Object[] { resultSet };
    }
    /**
     * Called when the procedure reference is no longer needed. Close
     * can be called without retrieving any of the output values (such
     * as cursors) or even invoking, so this needs to do any remaining
     * cleanup. Close can be called concurrently with any other call
     * such as "invoke" or "getOutputValues". In this case, any pending
     * methods should immediately throw a CustomProcedureException.
     */

    public void close() throws SQLException {
        if (resultSet != null) {
            resultSet.close();
        }
    }
    //
    // Introspection methods
    //
    /**
     * Called during introspection to get the short name of the stored
     * procedure. This name can be overridden during configuration.
     * Should not return null.
     */

    public String getName() {
        return "SimpleQuery";
    }
    /**

```

```

    * Called during introspection to get the description of the stored
    * procedure. Should not return null.
    */

    public String getDescription() {
        return "This procedure performs a simple query operation";
    }

//
// Transaction methods
//
/**
 * Returns true if the custom procedure uses transactions. If this
 * method returns false then commit and rollback will not be called.
 */

    public boolean canCommit() {

        return false;
    }
    /**
     * Commit any open transactions.
     */

    public void commit() { }

    /**
     * Rollback any open transactions.
     */

    public void rollback() { }
    /**
     * Returns true if the transaction can be compensated.
     */

    public boolean canCompensate() {
        return false;
    }
    /**
     * Compensate any committed transactions (if supported).
     */

    public void compensate(ExecutionEnvironment qenv) { }
}

```

Example 2: Simple Update

This custom procedure participates in the parent transaction, and performs an update using the execution environment.

```
package proc;
```

```

import com.compositesw.extension.*;
import java.sql.*;

public class SimpleUpdate

implements CustomProcedure
{

    private ExecutionEnvironment qenv;
    private int numRowsUpdated = -1;

    public SimpleUpdate() { }

    /**
     * This is called once just after constructing the class. The
     * environment contains methods used to interact with the server.
     */

    public void initialize(ExecutionEnvironment qenv) {
        this.qenv = qenv;
    }

    /**
     * Called during introspection to get the description of the input
     * and output parameters. Should not return null.
     */

    public ParameterInfo[] getParameterInfo() {

        return new ParameterInfo[] {
            new ParameterInfo("Id", Types.INTEGER, DIRECTION_IN),
            new ParameterInfo("FirstName", Types.VARCHAR, DIRECTION_IN),
            new ParameterInfo("LastName", Types.VARCHAR, DIRECTION_IN),
            new ParameterInfo("CompanyName", Types.VARCHAR, DIRECTION_IN),
            new ParameterInfo("PhoneNumber", Types.VARCHAR, DIRECTION_IN),
        };
    }

    /**
     * Called to invoke the stored procedure. Will only be called a
     * single time per instance. Can throw CustomProcedureException or
     * SQLException if there is an error during invoke.
     */

    public void invoke(Object[] inputValues)
        throws CustomProcedureException, SQLException
    {
        // Update in the first data source using a SQL statement
        numRowsUpdated = qenv.executeUpdate(
            "UPDATE /shared/tutorial/sources/ds_orders/customers" +
            " SET ContactFirstName='" + inputValues[1] +
            "', ContactLastName='" + inputValues[2] +
            "', CompanyName='" + inputValues[3] +
            "', PhoneNumber='" + inputValues[4] +
            "' WHERE CustomerID=" + inputValues[0],
            null);
    }
}

```

```

/**
 * Called to retrieve the number of rows that were inserted,
 * updated, or deleted during the execution of the procedure. A
 * return value of -1 indicates that the number of affected rows is
 * unknown. Can throw CustomProcedureException or SQLException if
 * there is an error when getting the number of affected rows.
 */

public int getNumAffectedRows() {
    return numRowsUpdated;
}

/**
 * Called to retrieve the output values. The returned objects
 * should obey the Java to SQL typing conventions as defined in the
 * table above. Output cursors can be returned as either
 * CustomCursor or java.sql.ResultSet. Can throw
 * CustomProcedureException or SQLException if there is an error
 * when getting the output values. Should not return null.
 */

public Object[] getOutputValues() {
    return new Object[] { };
}

/**
 * Called when the procedure reference is no longer needed. Close
 * can be called without retrieving any of the output values (such
 * as cursors) or even invoking, so this needs to do any remaining
 * cleanup. Close can be called concurrently with any other call
 * such as "invoke" or "getOutputValues". In this case, any pending
 * methods should immediately throw a CustomProcedureException.
 */

public void close() { }

//
// Introspection methods
//

/**
 * Called during introspection to get the short name of the stored
 * procedure. This name can be overridden during configuration.
 * Should not return null.
 */

public String getName() {
    return "SimpleUpdate";
}

/**
 * Called during introspection to get the description of the stored
 * procedure. Should not return null.
 */

public String getDescription() {
    return "This procedure performs a simple update operation";
}

```

```

//
// Transaction methods
//

/**
 * Returns true if the custom procedure uses transactions. If this
 * method returns false then commit and rollback will not be called.
 */

public boolean canCommit() {
    return false;
}

/**
 * Commit any open transactions.
 */

public void commit() { }

/**
 * Rollback any open transactions.
 */

public void rollback() { }

/**
 * Returns true if the transaction can be compensated.
 */

public boolean canCompensate() {
    return false;
}

/**
 * Compensate any committed transactions (if supported).
 */

public void compensate(ExecutionEnvironment qenv) { }
}

```

Example 3: External Update without Compensation

This custom procedure uses an independent transaction with a transactional data source in the server. Compensating logic is defined for the independent transaction.

```

package proc;

import com.compositesw.extension.*;
import java.sql.*;

```

```

public class ExternalUpdate
    implements CustomProcedure, java.io.Serializable
{
    private static final String ORDERS_URL =
        "jdbc:mysql://localhost:3306/Orders";
    private transient ExecutionEnvironment qenv;
    private transient Connection conn;
    private transient int numRowsUpdated;
    private boolean isUpdate;
    private int id;
    private String firstName;
    private String lastName;
    private String companyName;
    private String phoneNumber;

    public ExternalUpdate() { }

    /**
     * This is called once just after constructing the class. The
     * environment contains methods used to interact with the server.
     */

    public void initialize(ExecutionEnvironment qenv)
        throws SQLException
    {
        this.qenv = qenv;
        conn = DriverManager.getConnection(ORDERS_URL, "tutorial", "tutorial");
        conn.setAutoCommit(false);
    }

    /**
     * Called during introspection to get the description of the input
     * and output parameters. Should not return null.
     */

    public ParameterInfo[] getParameterInfo() {
        return new ParameterInfo[] {
            new ParameterInfo("Id", Types.INTEGER, DIRECTION_IN),
            new ParameterInfo("FirstName", Types.VARCHAR, DIRECTION_IN),
            new ParameterInfo("LastName", Types.VARCHAR, DIRECTION_IN),
            new ParameterInfo("CompanyName", Types.VARCHAR, DIRECTION_IN),
            new ParameterInfo("PhoneNumber", Types.VARCHAR, DIRECTION_IN),
        };
    }

    /**
     * Called to invoke the stored procedure. Will only be called a
     * single time per instance. Can throw CustomProcedureException or
     * SQLException if there is an error during invoke.
     */

    public void invoke(Object[] inputValues)
        throws CustomProcedureException, SQLException
    {
        Statement stmt = conn.createStatement();

```

```

//
// Save away the current values to be used for compensation
//

ResultSet rs = stmt.executeQuery(
    "SELECT ContactFirstName, ContactLastName, CompanyName, PhoneNumber " +
    "FROM customers WHERE CustomerID=" + inputValues[0]);
if (rs.next()) {
    isUpdate = true;
    id = ((Integer)inputValues[0]).intValue();
    firstName = rs.getString(1);
    lastName = rs.getString(2);
    companyName = rs.getString(3);
    phoneNumber = rs.getString(4);
}

rs.close();

//
// Perform the insert or update
//

if (isUpdate) {
    numRowsUpdated = stmt.executeUpdate(
        "UPDATE customers" +
        " SET ContactFirstName=" + inputValues[1] +
        "", ContactLastName=" + inputValues[2] +
        "", CompanyName=" + inputValues[3] +
        "", PhoneNumber=" + inputValues[4] +
        " WHERE CustomerID=" + inputValues[0]);
}
else {
    numRowsUpdated = stmt.executeUpdate(
        "INSERT into customers (CustomerID, ContactFirstName, " +
        "ContactLastName, CompanyName, PhoneNumber) VALUES (" +
        inputValues[0] + ", " + inputValues[1] + ", " +
        inputValues[2] + ", " + inputValues[3] + ", " +
        inputValues[4] + ")");
}
stmt.close();
}

/**
 * Called to retrieve the number of rows that were inserted,
 * updated, or deleted during the execution of the procedure. A
 * return value of -1 indicates that the number of affected rows is
 * unknown. Can throw CustomProcedureException or SQLException if
 * there is an error when getting the number of affected rows.
 */

public int getNumAffectedRows() {
    return numRowsUpdated;
}

/**
 * Called to retrieve the output values. The returned objects
 * should obey the Java to SQL typing conventions as defined in the

```

```

* table above. Output cursors can be returned as either
* CustomCursor or java.sql.ResultSet. Can throw
* CustomProcedureException or SQLException if there is an error
* when getting the output values. Should not return null.
*/

public Object[] getOutputValues() {
    return new Object[] { };
}

/**
 * Called when the procedure reference is no longer needed. Close
 * can be called without retrieving any of the output values (such
 * as cursors) or even invoking, so this needs to do any remaining
 * cleanup. Close can be called concurrently with any other call
 * such as "invoke" or "getOutputValues". In this case, any pending
 * methods should immediately throw a CustomProcedureException.
 */

public void close()
    throws SQLException
{ }

//
// Introspection methods
//

/**
 * Called during introspection to get the short name of the stored
 * procedure. This name can be overridden during configuration.
 * Should not return null.
 */

public String getName() {
    return "ExternalUpdate";
}

/**
 * Called during introspection to get the description of the stored
 * procedure. Should not return null.
 */

public String getDescription() {
    return "This procedure performs an update to an external transactional " +
        "data source using JDBC.";
}

//
// Transaction methods
//

/**
 * Returns true if the custom procedure uses transactions. If this
 * method returns false then commit and rollback will not be called.
 */

public boolean canCommit() {

```



```

        return true;
    }

    /**
     * Commit any open transactions
     */

    public void commit()
        throws SQLException
    {
        conn.commit();
        conn.close();
        conn = null;
    }

    /**
     * Rollback any open transactions.
     */

    public void rollback()
        throws SQLException
    {
        conn.rollback();
        conn.close();
        conn = null;
    }

    /**
     * Returns true if the transaction can be compensated.
     */

    public boolean canCompensate() {
        return true;
    }

    /**
     * Compensate any committed transactions (if supported).
     */

    public void compensate(ExecutionEnvironment qenv)
        throws SQLException
    {
        conn = DriverManager.getConnection(ORDERS_URL);
        conn.setAutoCommit(false);
        Statement stmt = conn.createStatement();
        if (isUpdate) {
            numRowsUpdated = stmt.executeUpdate(
                "UPDATE customers" +
                " SET ContactFirstName='" + firstName +
                "', ContactLastName='" + lastName +
                "', CompanyName='" + companyName +
                "', PhoneNumber='" + phoneNumber +
                " WHERE CustomerID=" + id);
        }

        else {

```

```

        stmt.executeUpdate("DELETE from customers WHERE CustomerID=" + id);
    }
    stmt.close();
    conn.commit();
    conn.close();
    conn = null;
}
}

```

Example 4: Nontransactional External Update without Compensation

This custom procedure updates the contents of a file on disk where the file is nontransactional. The actual work is deferred until the commit method is called. Compensating logic is provided.

```
package proc;
```

```

import com.compositesw.extension.*;
import java.sql.*;
import java.io.*;

public class NonTransactional
    implements CustomProcedure, java.io.Serializable
{
    private transient ExecutionEnvironment qenv;
    private transient File dataFile;
    private transient int numRowsUpdated;
    private transient int newId;
    private transient String newFirstName;
    private transient String newLastName;
    private transient String newCompanyName;
    private transient String newPhoneNumber;
    private int oldId;
    private String oldFirstName;
    private String oldLastName;
    private String oldCompanyName;
    private String oldPhoneNumber;

    public NonTransactional() { }

    /**
     * This is called once just after constructing the class. The
     * environment contains methods used to interact with the server.
     */

    public void initialize(ExecutionEnvironment qenv)
        throws CustomProcedureException
    {
        this.qenv = qenv;
    }
}

```

```

dataFile = new File("C:/CustomProcNonTrans.txt");
try {
    if (!dataFile.canWrite() && !dataFile.createNewFile())
        throw new CustomProcedureException("cannot write file");
}

catch (IOException ex) {
    throw new CustomProcedureException(ex);
}

/**
 * Called during introspection to get the description of the input
 * and output parameters. Should not return null.
 */

public ParameterInfo[] getParameterInfo() {
    return new ParameterInfo[] {
        new ParameterInfo("Id", Types.INTEGER, DIRECTION_IN),
        new ParameterInfo("FirstName", Types.VARCHAR, DIRECTION_IN),
        new ParameterInfo("LastName", Types.VARCHAR, DIRECTION_IN),
        new ParameterInfo("CompanyName", Types.VARCHAR, DIRECTION_IN),
        new ParameterInfo("PhoneNumber", Types.VARCHAR, DIRECTION_IN),
    };
}

/**
 * Called to invoke the stored procedure. Will only be called a
 * single time per instance. Can throw CustomProcedureException or
 * SQLException if there is an error during invoke.
 */

public void invoke(Object[] inputValues)
    throws CustomProcedureException
{
    //
    // Save new values for later use in 'commit'
    //

    newId = ((Integer)inputValues[0]).intValue();
    newFirstName = (String)inputValues[1];
    newLastName = (String)inputValues[2];
    newCompanyName = (String)inputValues[2];
    newPhoneNumber = (String)inputValues[3];
}

/**
 * Called to retrieve the number of rows that were inserted,
 * updated, or deleted during the execution of the procedure. A
 * return value of -1 indicates that the number of affected rows is
 * unknown. Can throw CustomProcedureException or SQLException if
 * there is an error when getting the number of affected rows.
 */

public int getNumAffectedRows()
    throws CustomProcedureException
{

```

```

        return numRowsUpdated;
    }

    /**
     * Called to retrieve the output values. The returned objects
     * should obey the Java to SQL typing conventions as defined in the
     * table above. Output cursors can be returned as either
     * CustomCursor or java.sql.ResultSet. Can throw
     * CustomProcedureException or SQLException if there is an error
     * when getting the output values. Should not return null.
     */

    public Object[] getOutputValues()
        throws CustomProcedureException
    {
        return new Object[] { };
    }

    /**
     * Called when the procedure reference is no longer needed. Close
     * can be called without retrieving any of the output values (such
     * as cursors) or even invoking, so this needs to do any remaining
     * cleanup. Close can be called concurrently with any other call
     * such as "invoke" or "getOutputValues". In this case, any pending
     * methods should immediately throw a CustomProcedureException.
     */

    public void close() { }

    //
    // Introspection methods
    //

    /**
     * Called during introspection to get the short name of the stored
     * procedure. This name can be overridden during configuration.
     * Should not return null.
     */

    public String getName() {
        return "NonTransactional";
    }

    /**
     * Called during introspection to get the description of the stored
     * procedure. Should not return null.
     */

    public String getDescription() {
        return "This procedure performs an update to an external " +
            "nontransactional file data source.";
    }

    //
    // Transaction methods
    //

```

```

/**
 * Returns true if the custom procedure uses transactions. If this
 * method returns false then commit and rollback will not be called.
 */

public boolean canCommit() {
    return true;
}

/**
 * Commit any open transactions.
 */

public void commit()
    throws CustomProcedureException
{
    //
    // Save away the current values to be used for compensation
    //
    try {
        BufferedReader reader = new BufferedReader(new FileReader(dataFile));
        String line = reader.readLine();
        oldId = (line == null || line.length() == 0) ? 0 : Integer.parseInt(line);
        oldFirstName = reader.readLine();
        oldLastName = reader.readLine();
        oldCompanyName = reader.readLine();
        oldPhoneNumber = reader.readLine();
        reader.close();
    }
    catch (IOException ex) {
        throw new CustomProcedureException(ex);
    }

    //
    // Write the new data out to the file
    //
    try {
        BufferedWriter writer = new BufferedWriter(new FileWriter(dataFile));
        writer.write(Integer.toString(newId)); writer.newLine();
        writer.write(newFirstName);          writer.newLine();
        writer.write(newLastName);            writer.newLine();
        writer.write(newCompanyName);         writer.newLine();
        writer.write(newPhoneNumber);         writer.newLine();
        writer.close();
    }
    catch (IOException ex) {
        throw new CustomProcedureException(ex);
    }
}

/**
 * Rollback any open transactions.
 */

public void rollback() {
    // do nothing

```

```

    }

    /**
     * Returns true if the transaction can be compensated.
     */

    public boolean canCompensate() {
        return true;
    }

    /**
     * Compensate any committed transactions (if supported).
     */

    public void compensate(ExecutionEnvironment qenv)
        throws CustomProcedureException
    {
        //
        // Restore the old data
        //
        try {
            BufferedWriter writer = new BufferedWriter(new FileWriter(dataFile));
            writer.write(Integer.toString(oldId)); writer.newLine();
            writer.write(oldFirstName);          writer.newLine();
            writer.write(oldLastName);           writer.newLine();
            writer.write(oldCompanyName);        writer.newLine();
            writer.write(oldPhoneNumber);         writer.newLine();
            writer.close();
        }
        catch (IOException ex) {
            throw new CustomProcedureException(ex);
        }
    }
}

```

Example 5: Expression Evaluator

This custom procedure evaluates simple expressions.

```

package proc;

import com.compositesw.extension.*;
import java.sql.SQLException;
import java.sql.Types;

/**
 * Custom procedure to evaluate simple expressions:
 *
 * ARG1 | ARG2
 * ARG1 if it is neither null nor 0, otherwise ARG2
 *
 * ARG1 & ARG2
 * ARG1 if neither argument is null or 0, otherwise 0
 */

```

```

* ARG1 < ARG2
* ARG1 is less than ARG2
*
* ARG1 <= ARG2
* ARG1 is less than or equal to ARG2
*
* ARG1 = ARG2
* ARG1 is equal to ARG2
*
* ARG1 != ARG2
* ARG1 is unequal to ARG2
*
* ARG1 >= ARG2
* ARG1 is greater than or equal to ARG2
*
* ARG1 > ARG2
* ARG1 is greater than ARG2
*
* ARG1 + ARG2
* arithmetic sum of ARG1 and ARG2
*
* ARG1 - ARG2
* arithmetic difference of ARG1 and ARG2
*
* ARG1 * ARG2
* arithmetic product of ARG1 and ARG2
*
* ARG1 / ARG2
* arithmetic quotient of ARG1 divided by ARG2
*
* ARG1 % ARG2
* arithmetic remainder of ARG1 divided by ARG2
*/

public class ExpressionEvaluator

    implements CustomProcedure
    {
        private ExecutionEnvironment qenv;
        private int result;
        public ExpressionEvaluator() { }
        /**
         * This is called once just after constructing the class. The
         * environment contains methods used to interact with the server.
         */

        public void initialize(ExecutionEnvironment qenv)
            throws SQLException
        {
            this.qenv = qenv;
        }

        /**
         * Called during introspection to get the description of the input
         * and output parameters. Should not return null.
         */

```

```

public ParameterInfo[] getParameterInfo() {
    return new ParameterInfo[] {
        new ParameterInfo("arg1", Types.INTEGER, DIRECTION_IN),
        new ParameterInfo("operator", Types.VARCHAR, DIRECTION_IN),
        new ParameterInfo("arg2", Types.INTEGER, DIRECTION_IN),
        new ParameterInfo("result", Types.INTEGER, DIRECTION_OUT),
    };
}

/**
 * Called to invoke the stored procedure. Will only be called a
 * single time per instance. Can throw CustomProcedureException or
 * SQLException if there is an error during invoke.
 */

public void invoke(Object[] inputValues)
    throws CustomProcedureException, SQLException
{
    int arg1 =
        (inputValues[0] != null ? ((Integer)inputValues[0]).intValue() : 0);
    String op = (String)inputValues[1];
    int arg2 =
        (inputValues[2] != null ? ((Integer)inputValues[2]).intValue() : 0);
    if (op.equals(""))
        result = (arg1 != 0) ? arg1 : arg2;
    else if (op.equals("&"))
        result = (arg1 != 0 && arg2 != 0) ? arg1 : 0;
    else if (op.equals("<"))
        result = (arg1 < arg2) ? 1 : 0;
    else if (op.equals("<="))
        result = (arg1 <= arg2) ? 1 : 0;
    else if (op.equals("="))
        result = (arg1 == arg2) ? 1 : 0;
    else if (op.equals("!="))
        result = (arg1 != arg2) ? 1 : 0;
    else if (op.equals(">="))
        result = (arg1 >= arg2) ? 1 : 0;
    else if (op.equals(">"))
        result = (arg1 > arg2) ? 1 : 0;
    else if (op.equals("+"))
        result = arg1 + arg2;
    else if (op.equals("-"))
        result = arg1 - arg2;
    else if (op.equals("*"))
        result = arg1 * arg2;
    else if (op.equals("/"))
        result = arg1 / arg2;
    else if (op.equals("%"))
        result = arg1 % arg2;
    else
        throw new CustomProcedureException("Unknown operator: " + op);
}

/**
 * Called to retrieve the number of rows that were inserted,
 * updated, or deleted during the execution of the procedure. A
 * return value of -1 indicates that the number of affected rows is

```



```

* unknown. Can throw CustomProcedureException or SQLException if
* there is an error when getting the number of affected rows.
*/

public int getNumAffectedRows() {
    return 0;
}

/**
 * Called to retrieve the output values. The returned objects
 * should obey the Java to SQL typing conventions as defined in the
 * table above. Output cursors can be returned as either
 * CustomCursor or java.sql.ResultSet. Can throw
 * CustomProcedureException or SQLException if there is an error
 * when getting the output values. Should not return null.
 */

public Object[] getOutputValues() {
    return new Object[] { new Integer(result) };
}

/**
 * Called when the procedure reference is no longer needed. Close
 * can be called without retrieving any of the output values (such
 * as cursors) or even invoking, so this needs to do any remaining
 * cleanup. Close can be called concurrently with any other call
 * such as "invoke" or "getOutputValues". In this case, any pending
 * methods should immediately throw a CustomProcedureException.
 */

public void close()
    throws SQLException
{ }

//
// Introspection methods
//
/**
 * Called during introspection to get the short name of the stored
 * procedure. This name can be overridden during configuration.
 * Should not return null.
 */

public String getName() {
    return "expr";
}

/**
 * Called during introspection to get the description of the stored
 * procedure. Should not return null.
 */

public String getDescription() {
    return "Custom procedure to evaluate simple expressions";
}

//
// Transaction methods

```

```

//

/**
 * Returns true if the custom procedure uses transactions. If this
 * method returns false then commit and rollback will not be called.
 */

public boolean canCommit() {
    return false;
}

/**
 * Commit any open transactions.
 */

public void commit()
    throws SQLException
{ }

/**
 * Rollback any open transactions.
 */

public void rollback()
    throws SQLException
{ }

/**
 * Returns true if the transaction can be compensated.
 */

public boolean canCompensate() {
    return false;
}

/**
 * Compensate any committed transactions (if supported).
 */

public void compensate(ExecutionEnvironment qenv)
    throws SQLException
{ }
}

```

Example 6: Output Cursor

This custom procedure invokes another procedure, and retrieves output values.

```

package proc;

import com.compositesw.extension.*;
import java.sql.SQLException;
import java.sql.Timestamp;
import java.sql.Types;

```

```

public class OutputCursor
    implements CustomProcedure, java.io.Serializable
{
    private transient ExecutionEnvironment qenv;
    private transient CustomCursor outputCursor;
    private boolean invoked;

    public OutputCursor() { }

    /**
     * This is called once just after constructing the class. The
     * environment contains methods used to interact with the server.
     */

    public void initialize(ExecutionEnvironment qenv)
        throws SQLException
    {
        this.qenv = qenv;
    }

    /**
     * Called during introspection to get the description of the input
     * and output parameters. Should not return null.
     */

    public ParameterInfo[] getParameterInfo() {
        return new ParameterInfo[] {
            new ParameterInfo("result", TYPED_CURSOR, DIRECTION_OUT,
                new ParameterInfo[] {
                    new ParameterInfo("IntColumn", Types.INTEGER, DIRECTION_NONE),
                    new ParameterInfo("StringColumn", Types.VARCHAR, DIRECTION_NONE),
                    new ParameterInfo("TimestampColumn", Types.TIMESTAMP, DIRECTION_NONE),
                })
        };
    }

    /**
     * Called to invoke the stored procedure. Will only be called a
     * single time per instance. Can throw CustomProcedureException or
     * SQLException if there is an error during invoke.
     */

    public void invoke(Object[] inputValues)
        throws CustomProcedureException, SQLException
    {
        invoked = true;
    }

    /**
     * Called to retrieve the number of rows that were inserted,
     * updated, or deleted during the execution of the procedure. A
     * return value of -1 indicates that the number of affected rows is
     * unknown. Can throw CustomProcedureException or SQLException if
     * there is an error when getting the number of affected rows.
     */

```

```

public int getNumAffectedRows() {
    return 0;
}

/**
 * Called to retrieve the output values. The returned objects
 * should obey the Java to SQL typing conventions as defined in the
 * table above. Output cursors can be returned as either
 * CustomCursor or java.sql.ResultSet. Can throw
 * CustomProcedureException or SQLException if there is an error
 * when getting the output values. Should not return null.
 */

public Object[] getOutputValues() {
    outputCursor = createCustomCursor();
    return new Object[] { outputCursor };
}

/**
 * Create a custom cursor output.
 */

private static CustomCursor createCustomCursor() {
    return new CustomCursor() {
        private int counter;
        public ParameterInfo[] getColumnInfo() {

            return null;
        }

        public Object[] next()
            throws CustomProcedureException, SQLException
        {
            if (counter++ >= 10) {
                return null;
            }
            else {
                return new Object[] {
                    new Integer(counter),
                    Integer.toString(counter),
                    new Timestamp(counter),
                };
            }
        }

        public void close()
            throws CustomProcedureException, SQLException
        {
            // do nothing
        }
    };
}

/**
 * Called when the procedure reference is no longer needed. Close
 * can be called without retrieving any of the output values (such
 * as cursors) or even invoking, so this needs to do any remaining

```

```

    * cleanup. Close can be called concurrently with any other call
    * such as "invoke" or "getOutputValues". In this case, any pending
    * methods should immediately throw a CustomProcedureException.
    */

    public void close()
        throws CustomProcedureException, SQLException
    {
        if (outputCursor != null)
            outputCursor.close();
    }
    //
    // Introspection methods
    //

    /**
     * Called during introspection to get the short name of the stored
     * procedure. This name can be overridden during configuration.
     * Should not return null.
     */

    public String getName() {
        return "OutputCursor";
    }

    /**
     * Called during introspection to get the description of the stored
     * procedure. Should not return null.
     */

    public String getDescription() {
        return "Custom procedure that returns cursor data";
    }
    //
    // Transaction methods
    //

    /**
     * Returns true if the custom procedure uses transactions. If this
     * method returns false then commit and rollback will not be called.
     */

    public boolean canCommit() {
        return true;
    }

    /**
     * Commit any open transactions.
     */

    public void commit()
        throws SQLException
    { }

    /**
     * Rollback any open transactions.

```

```

    */

    public void rollback()
        throws SQLException
    { }

    /**
     * Returns true if the transaction can be compensated.
     */

    public boolean canCompensate() {
        return true;
    }

    /**
     * Compensate any committed transactions (if supported).
     */

    public void compensate(ExecutionEnvironment qenv)
        throws SQLException
    {
        System.out.println("OutputCursor.compensate(): invoked=" + invoked);
    }
}

```

Example 7: Simple Procedure that Invokes Another Procedure

This custom procedure invokes another procedure.

```

package proc;

import com.compositesw.extension.*;
import java.sql.*;

public class SimpleProcInvoke
    implements CustomProcedure
{
    private ExecutionEnvironment qenv;
    private ProcedureReference proc;

    public SimpleProcInvoke() { }

    /**
     * This is called once just after constructing the class. The
     * environment contains methods used to interact with the server.
     */

    public void initialize(ExecutionEnvironment qenv) {
        this.qenv = qenv;
    }

    /**
     * Called during introspection to get the description of the input
     * and output parameters. Should not return null.
     */
}

```

```

*/

public ParameterInfo[] getParameterInfo() {
    return new ParameterInfo[] {
        new ParameterInfo("arg1", Types.INTEGER, DIRECTION_IN),
        new ParameterInfo("operator", Types.VARCHAR, DIRECTION_IN),
        new ParameterInfo("arg2", Types.INTEGER, DIRECTION_IN),
        new ParameterInfo("result", Types.INTEGER, DIRECTION_OUT),
    };
}

/**
 * Called to invoke the stored procedure. Will only be called a
 * single time per instance. Can throw CustomProcedureException or
 * SQLException if there is an error during invoke.
 */

public void invoke(Object[] inputValues)
    throws CustomProcedureException, SQLException
{
    proc = qenv.lookupProcedure("/services/databases/tutorial/expr");
    proc.invoke(inputValues);
}

/**
 * Called to retrieve the number of rows that were inserted,
 * updated, or deleted during the execution of the procedure. A
 * return value of -1 indicates that the number of affected rows is
 * unknown. Can throw CustomProcedureException or SQLException if
 * there is an error when getting the number of affected rows.
 */

public int getNumAffectedRows() {
    return 0;
}

/**
 * Called to retrieve the output values. The returned objects
 * should obey the Java to SQL typing conventions as defined in the
 * table above. Output cursors can be returned as either
 * CustomCursor or java.sql.ResultSet. Can throw
 * CustomProcedureException or SQLException if there is an error
 * when getting the output values. Should not return null.
 */

public Object[] getOutputValues()
    throws CustomProcedureException, SQLException
{
    return proc.getOutputValues();
}

/**
 * Called when the procedure reference is no longer needed. Close
 * can be called without retrieving any of the output values (such
 * as cursors) or even invoking, so this needs to do any remaining
 * cleanup. Close can be called concurrently with any other call
 * such as "invoke" or "getOutputValues". In this case, any pending

```

```

    * methods should immediately throw a CustomProcedureException.
    */

    public void close()
        throws CustomProcedureException, SQLException
    {
        if (proc != null)
            proc.close();
    }

    //
    // Introspection methods
    //

    /**
     * Called during introspection to get the short name of the stored
     * procedure. This name can be overridden during configuration.
     * Should not return null.
     */

    public String getName() {
        return "SimpleProcInvoke";
    }

    /**
     * Called during introspection to get the description of the stored
     * procedure. Should not return null.
     */

    public String getDescription() {
        return "This procedure invokes another procedure.";
    }

    //
    // Transaction methods
    //

    /**
     * Returns true if the custom procedure uses transactions. If this
     * method returns false then commit and rollback will not be called.
     */

    public boolean canCommit() {
        return false;
    }

    /**
     * Commit any open transactions.
     */

    public void commit() { }

    /**
     * Rollback any open transactions.
     */

    public void rollback() { }

```



```
/**
 * Returns true if the transaction can be compensated.
 */

public boolean canCompensate() {
    return false;
}

/**
 * Compensate any committed transactions (if supported).
 */

public void compensate(ExecutionEnvironment qenv) { }
```


Function Support Summary

This topic contains an alphabetical list of functions supported in the TDV Server (first column of table). The second column indicates whether the function is supported in the TDV repository, The third column indicates whether the function is supported in data sources, including TDV *as a data source*. The final column indicates functions that are push-only for some or all of the supported data sources.

For details on specific data source *versions* that support individual functions, refer to [Function Support for Data Sources, page 519](#)

| Function Name | TDV Repository | Supporting Data Source | Push - only |
|---------------|----------------|--|-------------|
| ABS | Yes | TDV, DataDirect Mainframe, DB2 DataDirect Mainframe, DB2, DB2 Mainframe, File, Greenplum, HBASE, Hive, HSQLDB, Impala, Informix, Microsoft Access, Microsoft SQL Server, MySQL, Neoview, Netezza, Oracle, PostgreSQL, SAP HANA, Sybase, Sybase IQ, Teradata, Vertica, XML File | |
| ACOS | Yes | TDV, DataDirect Mainframe, DB2 DataDirect Mainframe, DB2, DB2 Mainframe, File, Greenplum, Hive, HSQLDB, Impala, Informix, Microsoft SQL Server, MySQL, Neoview, Netezza, Oracle, PostgreSQL, SAP HANA, Sybase, Sybase IQ, Teradata, Vertica, XML File | |
| ADD_DAYS | | SAP HANA | |
| ADD_MONTH | | SAP HANA | |
| ADD_MONTHS | | Netezza, Oracle, Teradata, Vertica | |
| ASCII | | Hive, HSQLDB, Impala, Microsoft SQL Server, Netezza, Oracle, PostgreSQL, SAP HANA, Teradata v15, Vertica | |
| ASIN | Yes | TDV, DataDirect Mainframe, DB2 DataDirect Mainframe, DB2, DB2 Mainframe, File, Greenplum, Hive, HSQLDB, Impala, Informix, Microsoft SQL Server, MySQL, Neoview, Netezza, Oracle, PostgreSQL, Sybase, Sybase IQ, Teradata, Vertica, XML File | |

| Function Name | TDV Repository | Supporting Data Source | Push - only |
|---------------------|----------------|---|-------------|
| AT TIME ZONE | | | |
| ATAN | Yes | TDV, DataDirect Mainframe, DB2 DataDirect Mainframe, DB2, DB2 Mainframe, File, Greenplum, Hive, HSQLDB, Impala, Informix, Microsoft Access, Microsoft SQL Server, MySQL, Neoview, Netezza, Oracle, PostgreSQL, SAP HANA, Sybase, Sybase IQ, Teradata, Vertica, XML File | |
| ATAN2 | | Netezza, Oracle, PostgreSQL, SAP HANA | |
| AVG | Yes | TDV, DataDirect Mainframe, DB2 DataDirect Mainframe, DB2, DB2 Mainframe, File, Greenplum, Hive, HSQLDB, Impala, Informix, JDBC, Microsoft Access, Microsoft SQL Server, MySQL, Neoview, Netezza, Oracle, PostgreSQL, SAP HANA, Sybase, Sybase IQ, Teradata, Vertica, Vertica OLAP, XML File | |
| AVG (analytical) | | TDV, DB2, Greenplum, Microsoft SQL Server, Oracle, SAP HANA, Sybase IQ, Teradata, Vertica | |
| BINTOHEX | | SAP HANA | |
| BIT_AND | | Vertica | Yes |
| BIT_LENGTH | | Oracle, Vertica | Vertica |
| BIT_OR | | Vertica | Yes |
| BIT_XOR | | Vertica | Yes |
| BITAND | | SAP HANA | |
| BITCOUNT | | SAP HANA, Vertica | Yes |
| BITNOT | | SAP HANA | |
| BITOR | | SAP HANA | |
| BITSTRING_TO_BINARY | | Vertica | Yes |

| Function Name | TDV Repository | Supporting Data Source | Push - only |
|---------------------|----------------|--|-------------|
| BITXOR | | SAP HANA | |
| BTRIM | | Netezza, Oracle, PostgreSQL, Vertica | |
| CAST | Yes | TDV, DataDirect Mainframe, DB2 DataDirect Mainframe, DB2, DB2 Mainframe, File, Greenplum, HBASE, Hive, HSQLDB, Impala, Informix, Microsoft Access, Microsoft SQL Server, MySQL, Neoview, Netezza, Oracle, PostgreSQL, SAP HANA, Sybase, Sybase IQ, Teradata, Vertica, XML File | |
| CBRT | | Vertica | |
| CEIL | Yes | HBASE, Hive, HSQLDB, Impala, Netezza, Oracle, PostgreSQL, SAP HANA | |
| CEILING | Yes | TDV, DataDirect Mainframe, DB2 DataDirect Mainframe, DB2, DB2 Mainframe, File, Greenplum, HBASE, Hive, HSQLDB, Impala, Microsoft SQL Server, MySQL, Neoview, Netezza, Oracle, PostgreSQL, Sybase, Sybase IQ, Teradata, Vertica, XML File | |
| CHAR | | SAP HANA | |
| CHAR_LENGTH | | Greenplum, Teradata, Vertica | |
| CHARACTER_LENGTH | | Vertica | |
| CHR | | Netezza, Oracle, PostgreSQL, Teradata v15, Vertica | |
| CLOCK_TIME STAMP | | PostgreSQL, Vertica | Vertica |
| COALESCE | | DB2, HBASE, Hive, HSQLDB, Impala, Microsoft SQL Server, MySQL, Netezza, Oracle, Sybase, Sybase IQ, Teradata | |

| Function Name | TDV Repository | Supporting Data Source | Push - only |
|----------------------------|----------------|--|-------------|
| CONCAT | Yes | TDV, DataDirect Mainframe, DB2 DataDirect Mainframe, DB2, DB2 Mainframe, File, Greenplum, HBASE, Hive, HSQLDB, Impala, Informix, Microsoft Access, Microsoft SQL Server, MySQL, Neoview, Netezza, Oracle, PostgreSQL, SAP HANA, Sybase, Sybase IQ, Teradata, Vertica, XML File | |
| CONDITIONAL_CHANGE_EVENT | | TDV, Vertica | |
| CONDITIONAL_TRUE_EVENT | | Vertica | Yes |
| CORR | | DB2, Greenplum, Hive, HSQLDB, Impala, Oracle, PostgreSQL, SAP HANA, Sybase IQ, Vertica | |
| CORR (analytical) | | TDV, Greenplum, Oracle, SAP HANA, Vertica | |
| CORR_SPEARMAN | | SAP HANA | |
| CORR_SPEARMAN (analytical) | | SAP HANA | |
| COS | Yes | TDV, DataDirect Mainframe, DB2 DataDirect Mainframe, DB2, DB2 Mainframe, File, Greenplum, Hive, HSQLDB, Impala, Informix, Microsoft Access, Microsoft SQL Server, MySQL, Neoview, Netezza, Oracle, PostgreSQL, SAP HANA, Sybase, Sybase IQ, Teradata, Vertica, XML File | |
| COT | Yes | TDV, DataDirect Mainframe, DB2 DataDirect Mainframe, DB2, DB2 Mainframe, File, Greenplum, Microsoft Access, Microsoft SQL Server, MySQL, Neoview, Netezza, Oracle, PostgreSQL, SAP HANA, Sybase, Sybase IQ, Teradata, Vertica, XML File | |

| Function Name | TDV Repository | Supporting Data Source | Push - only |
|-------------------------|----------------|--|-------------|
| COUNT | Yes | TDV, DataDirect Mainframe, DB2 DataDirect Mainframe, DB2, DB2 Mainframe, File, Greenplum, HBASE, Hive, HSQLDB, Impala, Informix, JDBC, Microsoft Access, Microsoft SQL Server, MySQL, Neoview, Netezza, Oracle, PostgreSQL, SAP HANA, Sybase, Sybase IQ, Teradata, Vertica, Vertica OLAP, XML File | |
| COUNT (analytical) | | TDV, DB2, Greenplum, Microsoft SQL Server, Oracle, (analytical), Sybase IQ, Teradata, Vertica | |
| COVAR_POP | | TDV, Greenplum, Hive, HSQLDB, Impala, Oracle, PostgreSQL, Sybase IQ, Vertica | |
| COVAR_POP (analytical) | | TDV, Greenplum, Oracle, Vertica | |
| COVAR_SAMP | | TDV, Greenplum, Hive, HSQLDB, Impala, Oracle, PostgreSQL, Sybase IQ, Vertica | |
| COVAR_SAMP (analytical) | | TDV, Greenplum, Oracle, Vertica | |
| CUME | | Microsoft SQL Server | |
| CUME_DIST | | TDV, Greenplum, Oracle, PostgreSQL, SAP HANA, Sybase IQ, Teradata v15, Vertica, Vertica OLAP | |
| CURRENT_DATE | Yes | DataDirect Mainframe, DB2 DataDirect Mainframe, DB2, DB2 Mainframe, Greenplum, HBASE, Hive, HSQLDB, Impala, Informix, Microsoft Access, Microsoft SQL Server, MySQL, Neoview, Netezza, Oracle, PostgreSQL, SAP HANA, Sybase, Teradata, Vertica | |
| CURRENT_TIME | Yes | DataDirect Mainframe, DB2 DataDirect Mainframe, DB2, DB2 Mainframe, Greenplum, HBASE, Hive, HSQLDB, Impala, Microsoft Access, MySQL, Neoview, Netezza, SAP HANA, Sybase, Teradata, Vertica | |

| Function Name | TDV Repository | Supporting Data Source | Push - only |
|----------------------|----------------|--|-------------|
| CURRENT_TIMESTAMP | Yes | DataDirect Mainframe, DB2 DataDirect Mainframe, DB2, DB2 Mainframe, Greenplum, HSQLDB, Impala, Microsoft Access, Microsoft SQL Server, MySQL, Neoview, Netezza, Oracle, PostgreSQL, SAP HANA, Sybase, Sybase IQ, Teradata, Vertica | |
| CURRENT_UTCTIMESTAMP | | SAP HANA | |
| DATE_ADD | | Hive, HSQLDB, Impala | |
| DATE_PART | | Netezza, Vertica | |
| DATE_SUB | | Hive, HSQLDB, Impala | |
| DATE_TRUNC | | Netezza, Oracle, PostgreSQL, Vertica | |
| DATEADD | | Microsoft SQL Server | |
| DATEDIFF | | Hive, HSQLDB, Impala, Vertica | |
| DATENAME | | Microsoft SQL Server | |
| DATEPART | | Microsoft SQL Server | |
| DAY | Yes | DataDirect Mainframe, DB2 DataDirect Mainframe, DB2, DB2 Mainframe, File, Greenplum, Hive, HSQLDB, Impala, Informix, Microsoft Access, Microsoft SQL Server, MySQL, Neoview, Netezza, Oracle, PostgreSQL, Sybase, Sybase IQ, Teradata, Vertica, XML File | |
| DAYNAME | | Microsoft SQL Server, SAP HANA | |
| DAYOFMONTH | | TDV, Microsoft SQL Server, SAP HANA, Vertica | |
| DAYOFWEEK | | TDV, Microsoft SQL Server, Vertica | |
| DAYOFWEEK_ISO | | TDV, Vertica | |

| Function Name | TDV Repository | Supporting Data Source | Push - only |
|----------------------------|----------------|---|-------------|
| DAYOFYEAR | | TDV, SAP HANA, Vertica | |
| DAYS | | TDV, Microsoft Excel, Vertica* * Excel DAYS function is far different from Vertica DAYS function. | |
| DAYS_BETWEEN | | Oracle, SAP HANA | |
| DBL_MP | | Netezza | |
| DBTIMEZONE | | Teradata | |
| DECODE | | DB2, Microsoft SQL Server, MySQL, Neoview, Netezza, Oracle, Sybase, Sybase IQ, Teradata v15, Vertica | |
| DEGREES | Yes | TDV, DataDirect Mainframe, DB2 DataDirect Mainframe, DB2, DB2 Mainframe, File, Greenplum, HSQLDB, Microsoft SQL Server, MySQL, Neoview, Netezza, PostgreSQL, Sybase, Sybase IQ, Vertica, XML File | |
| DENSE_RANK | | TDV, DB2, Greenplum, Microsoft SQL Server, Netezza, Oracle, PostgreSQL, SAP HANA, Sybase IQ, Teradata v15, Vertica, Vertica OLAP | |
| DLE_DST | | Netezza | |
| EXP | Yes | TDV, DataDirect Mainframe, DB2 DataDirect Mainframe, DB2, DB2 Mainframe, File, Greenplum, Hive, HSQLDB, Impala, Informix, Microsoft Access, Microsoft SQL Server, MySQL, Neoview, Netezza, Oracle, PostgreSQL, SAP HANA, Sybase, Sybase IQ, Teradata, Vertica, XML File | |
| EXP_WEIGHTED_AVG | | Sybase IQ | Yes |
| EXPONENTIAL_MOVING_AVERAGE | | Vertica, Vertica OLAP | Yes |
| EXTRACT | | Greenplum, Hive, HSQLDB, Impala, Microsoft SQL Server, MySQL, Netezza, Oracle, PostgreSQL, SAP HANA, Vertica | |

| Function Name | TDV Repository | Supporting Data Source | Push - only |
|--------------------|----------------|------------------------|-------------|
| EXTRACTDAY | | Teradata | |
| EXTRACTDOW | | | |
| EXTRACTDOY | | | |
| EXTRACTEPOCH | | | |
| EXTRACTHOUR | | Teradata | |
| EXTRACTMICROSECOND | | | |
| EXTRACTMILLISECOND | | | |
| EXTRACTMINUTE | | Teradata | |
| EXTRACTMONTH | | Teradata | |
| EXTRACTQUARTER | | | |
| EXTRACTSECOND | | Teradata | |
| EXTRACTWEEK | | | |
| EXTRACTYEAR | | Teradata | |
| FACTORIAL | | Netezza | |
| FIND | | | |

| Function Name | TDV Repository | Supporting Data Source | Push - only |
|-----------------|----------------|--|-------------|
| FIND_IN_SET | | Hive, HSQLDB, Impala | Yes |
| FIRST_VALUE | | TDV, Greenplum, Microsoft SQL Server, Netezza, Oracle, SAP HANA, Sybase IQ, Teradata v15, Vertica, Vertica OLAP | |
| FLOOR | Yes | TDV, DataDirect Mainframe, DB2 DataDirect Mainframe, DB2, DB2 Mainframe, File, Greenplum, HBASE, Hive, HSQLDB, Impala, Microsoft SQL Server, MySQL, Neoview, Netezza, Oracle, PostgreSQL, SAP HANA, Sybase, Sybase IQ, Teradata, Vertica, XML File | |
| FORMAT_DATE | | DataDirect Mainframe, DB2 DataDirect Mainframe, DB2, Microsoft SQL Server, Oracle, PostgreSQL, Sybase, Sybase IQ, Teradata | |
| FROM_UNIXTIME | | Hive, HSQLDB, Impala | |
| GET_JSON_OBJECT | | TDV, Hive, HSQLDB, Impala | |
| GETUTCDATE | | TDV, Vertica | |
| GREATEST | | Oracle, SAP HANA, Teradata v15, Vertica | |
| HASHMD2 | | TDV, Microsoft SQL Server, Oracle | |
| HASHMD4 | | TDV, Microsoft SQL Server, Oracle | |
| HASHSHA | | TDV, Microsoft SQL Server, Oracle | |
| HASHSHA1 | | TDV, Microsoft SQL Server, Oracle | |
| HEX_TO_BINARY | | TDV, Vertica | |
| HEXTOBIN | | SAP HANA | |
| hour | | Microsoft SQL Server, SAP HANA, Vertica | |
| IFNULL | | TDV, SAP HANA, Vertica | |

| Function Name | TDV Repository | Supporting Data Source | Push - only |
|---------------|----------------|--|-------------|
| INET_ATON | | TDV, Vertica | |
| INET_NTOA | | TDV, Vertica | |
| INITCAP | | Netezza, Oracle, PostgreSQL, Teradata v15, Vertica | |
| INSTR | | Netezza, Oracle, Teradata v15, Vertica | |
| INT1AND | | Hive, HSQLDB, Impala, Netezza, Oracle, PostgreSQL, Vertica | |
| INT1NOT | | Hive, HSQLDB, Impala, Netezza, PostgreSQL, Vertica | |
| INT1OR | | Hive, HSQLDB, Impala, Netezza, PostgreSQL, Vertica | |
| INT1SHL | | Netezza | |
| INT1SHR | | Netezza | |
| INT1XOR | | Hive, HSQLDB, Impala, Netezza, PostgreSQL, Vertica | |
| INT2AND | | Hive, HSQLDB, Impala, Netezza, Oracle, PostgreSQL, Vertica | |
| INT2NOT | | Hive, HSQLDB, Impala, Netezza, PostgreSQL, Vertica | |
| INT2OR | | Hive, HSQLDB, Impala, Netezza, PostgreSQL, Vertica | |
| INT2SHL | | Netezza | |
| INT2SHR | | Netezza | |
| INT2XOR | | Hive, HSQLDB, Impala, Netezza, PostgreSQL, Vertica | |
| INT4AND | | Hive, HSQLDB, Impala, Netezza, Oracle, PostgreSQL, Vertica | |
| INT4NOT | | Hive, HSQLDB, Impala, Netezza, PostgreSQL, Vertica | |
| INT4OR | | Hive, HSQLDB, Impala, Netezza, PostgreSQL, Vertica | |
| INT4SHL | | Netezza | |
| INT4SHR | | Netezza | |

| Function Name | TDV Repository | Supporting Data Source | Push - only |
|---------------|----------------|---|-------------|
| INT4XOR | | Hive, HSQLDB, Impala, Netezza, PostgreSQL, Vertica | |
| INT8AND | | Hive, HSQLDB, Impala, Netezza, Oracle, PostgreSQL, Vertica | |
| INT8NOT | | Hive, HSQLDB, Impala, Netezza, PostgreSQL, Vertica | |
| INT8OR | | Hive, HSQLDB, Impala, Netezza, PostgreSQL, Vertica | |
| INT8SHL | | Netezza, Vertica | |
| INT8SHR | | Netezza, Vertica | |
| INT8XOR | | Hive, HSQLDB, Impala, Netezza, PostgreSQL, Vertica | |
| ISFINITE | | TDV, Vertica | |
| ISNULL | | Microsoft SQL Server, Netezza | |
| ISNUMERIC | | Microsoft SQL Server | |
| ISUTF8 | | TDV, Vertica | |
| JSONPATH | | | |
| JULIAN_DAY | | TDV, Vertica | |
| LAG | | TDV, Greenplum, Microsoft SQL Server, Netezza, Oracle, SAP HANA, Sybase IQ, Vertica, Vertica OLAP | |
| LAST_DAY | | Netezza, Oracle, SAP HANA, Teradata v15, Vertica | |
| LAST_VALUE | | TDV, Greenplum, Microsoft SQL Server, Netezza, Oracle, SAP HANA, Sybase IQ, Teradata v15, Vertica, Vertica OLAP | |
| LCASE | | TDV, HBASE, Hive, HSQLDB, Impala, SAP HANA | |
| LE_DST | | Netezza | |
| LEAD | | TDV, Greenplum, Microsoft SQL Server, Netezza, Oracle, SAP HANA, Sybase IQ, Teradata v15, Vertica, Vertica OLAP | |

| Function Name | TDV Repository | Supporting Data Source | Push - only |
|---------------|----------------|--|-------------|
| LEAST | | Oracle, SAP HANA, Teradata v15, Vertica | |
| LEFT | | SAP HANA, Teradata v15, Vertica | |
| LENGTH | Yes | TDV, DataDirect Mainframe, DB2 DataDirect Mainframe, DB2, DB2 Mainframe, File, Greenplum, HBASE, Hive, HSQLDB, Impala, Informix, Microsoft Access, Microsoft SQL Server, MySQL, Neoview, Netezza, Oracle, PostgreSQL, SAP HANA, Sybase, Sybase IQ, Teradata, Vertica, XML File | |
| LIKE_REGEX | | SAP HANA | |
| LISTAGG | | TDV, Oracle | |
| LN | | Hive, HSQLDB, Impala, Netezza, Oracle, SAP HANA | |
| LOCATE | | SAP HANA | |
| LOG | Yes | TDV, DataDirect Mainframe, DB2 DataDirect Mainframe, DB2, DB2 Mainframe, File, Greenplum, Hive, HSQLDB, Impala, Informix, Microsoft Access, Microsoft SQL Server, MySQL, Neoview, Netezza, Oracle, PostgreSQL, SAP HANA, Sybase, Sybase IQ, Teradata, Vertica, XML File | |
| LOWER | Yes | TDV, DataDirect Mainframe, DB2 DataDirect Mainframe, DB2, DB2 Mainframe, File, Greenplum, HBASE, Hive, HSQLDB, Impala, Informix, Microsoft Access, Microsoft SQL Server, MySQL, Neoview, Netezza, Oracle, PostgreSQL, SAP HANA, Sybase, Sybase IQ, Teradata, Vertica, XML File | |
| LPAD | | Hive, HSQLDB, Impala, Netezza, Oracle, PostgreSQL, SAP HANA, Teradata v15, Vertica | |
| LTRIM | | Greenplum, HBASE, Hive, HSQLDB, Impala, Netezza, Oracle, PostgreSQL, SAP HANA, Teradata v15, Vertica | |

| Function Name | TDV Repository | Supporting Data Source | Push - only |
|---------------------|----------------|--|-------------|
| MAX | Yes | TDV, DataDirect Mainframe, DB2 DataDirect Mainframe, DB2, DB2 Mainframe, File, Greenplum, HBASE, Hive, HSQLDB, Impala, Informix, JDBC, Microsoft Access, Microsoft SQL Server, MySQL, Neoview, Netezza, Oracle, PostgreSQL, SAP HANA, Sybase, Sybase IQ, Teradata, Vertica, Vertica OLAP, XML File | |
| MAX (analytical) | | TDV, DB2, Greenplum, Microsoft SQL Server, Oracle, SAP HANA, Teradata, Vertica | |
| MD5 | | TDV, Vertica | |
| MEDIAN | | TDV, Oracle, SAP HANA, Sybase, Teradata v15, Vertica OLAP | |
| MEDIAN (analytical) | | Oracle, SAP HANA, Vertica, Vertica OLAP | |
| MICROSECOND | | TDV, Vertica | |
| MIDNIGHT_SECONDS | | TDV, Vertica | |
| MIN | Yes | TDV, DataDirect Mainframe, DB2 DataDirect Mainframe, DB2, DB2 Mainframe, File, Greenplum, HBASE, Hive, HSQLDB, Impala, Informix, JDBC, Microsoft Access, Microsoft SQL Server, MySQL, Neoview, Netezza, Oracle, PostgreSQL, SAP HANA, Sybase, Sybase IQ, Teradata, Vertica, Vertica OLAP, XML File | |
| MIN (analytical) | | TDV, DB2, Greenplum, Microsoft SQL Server, Oracle, SAP HANA, Teradata, Vertica | |
| MINUTE | | Impala, Microsoft SQL Server, SAP HANA, Vertica | |
| MOD | | Oracle, PostgreSQL, SAP HANA, Vertica | |
| MONTH | Yes | DataDirect Mainframe, DB2 DataDirect Mainframe, DB2, DB2 Mainframe, File, Greenplum, Hive, HSQLDB, Impala, Informix, Microsoft Access, Microsoft SQL Server, MySQL, Neoview, Netezza, Oracle, PostgreSQL, SAP HANA, Sybase, Sybase IQ, Teradata, Vertica, XML File | |

| Function Name | TDV Repository | Supporting Data Source | Push - only |
|-------------------|----------------|--|-------------|
| MONTHNAME | | Microsoft SQL Server, SAP HANA | |
| MONTHS_BETWEEN | | Netezza, Oracle, Teradata v15, Vertica | |
| NCHAR | | SAP HANA | |
| NEW_TIME | | Oracle | |
| NEXT_DAY | | Netezza, Oracle, Vertica | |
| NOW | | Hive, HSQLDB, Impala, Netezza, Oracle, PostgreSQL, SAP HANA, Teradata v15, Vertica | |
| NTH_VALUE | | TDV, | |
| NTILE | | TDV, Greenplum, Microsoft SQL Server, Oracle, PostgreSQL, SAP HANA, Sybase IQ, Vertica, Vertica OLAP | |
| NULLIF | | TDV, DataDirect Mainframe, DB2 DataDirect Mainframe, DB2, DB2 Mainframe, Greenplum, Informix, Microsoft Access, Microsoft SQL Server, Neoview, Netezza, Oracle, SAP HANA, Sybase, Sybase IQ, Teradata, Vertica | |
| NUMERIC_LOG | | Netezza | |
| NUMTODSINTERVAL | | TDV, Oracle 9 or later | |
| NUMTOYMININTERVAL | | TDV, Oracle 9 or later | |
| NVL | | Greenplum, Microsoft SQL Server, Netezza, Oracle, Teradata, Vertica | |
| NVL2 | | Greenplum, Netezza, Oracle, Teradata v15, Vertica | |
| NYSIIS | | Netezza | |
| OCTET_LENGTH | | TDV, Vertica | |

| Function Name | TDV Repository | Supporting Data Source | Push - only |
|-------------------------------------|----------------|---|-------------|
| OVERLAYB | | TDV, Vertica | |
| PARSE_DATE | | Teradata | |
| PARSE_TIME | | Teradata | |
| PARSE_TIMES TAMP | | DataDirect Mainframe, DB2 DataDirect Mainframe, DB2, Greenplum, Microsoft SQL Server, Oracle, PostgreSQL, Sybase, Sybase IQ, Teradata | |
| PARSE_URL | | TDV, Hive, HSQLDB, Impala | |
| PARTIAL_STR ING_MASK | | | |
| PERCENT_RANK | | TDV, Greenplum, Oracle, PostgreSQL, SAP HANA, Sybase IQ, Teradata, Vertica, Vertica OLAP | |
| PERCENTILE | | Hive, HSQLDB, Impala, Oracle, Sybase IQ, Vertica | Yes |
| PERCENTILE_ APPROX | | Hive, HSQLDB, Impala | Yes |
| PERCENTILE_ CONT | | TDV, Oracle, Sybase IQ, Vertica | |
| PERCENTILE_ CONT (analytical) | | TDV, Microsoft SQL Server, Oracle, SAP HANA, Teradata v15, Vertica, Vertica OLAP | |
| PERCENTILE_ DISC | | TDV, Oracle, Sybase IQ, Vertica | |
| PERCENTILE_ DISC (analytical) | | TDV, Microsoft SQL Server, Oracle, SAP HANA, Teradata v15, Vertica, Vertica OLAP | |
| PI | | TDV, File, Microsoft SQL Server, Neoview, Netezza, PostgreSQL, Vertica, XML File | |

| Function Name | TDV Repository | Supporting Data Source | Push - only |
|------------------|----------------|---|-------------|
| POSITION | | TDV, DataDirect Mainframe, DB2 DataDirect Mainframe, DB2, DB2 Mainframe, Greenplum, HSQLDB, Microsoft SQL Server, MySQL, Neoview, Netezza, Oracle, Sybase, Sybase IQ, Teradata, Vertica | |
| POW | Yes | Hive, HSQLDB, Impala, Netezza, Oracle, PostgreSQL | |
| POWER | Yes | TDV, DataDirect Mainframe, DB2 DataDirect Mainframe, DB2, DB2 Mainframe, File, Greenplum, Hive, HSQLDB, Impala, Informix, Microsoft SQL Server, MySQL, Neoview, Netezza, Oracle, PostgreSQL, SAP HANA, Sybase, Sybase IQ, Vertica, XML File | |
| PRI_MP | | Netezza | |
| QUARTER | | TDV, Microsoft SQL Server, SAP HANA, Vertica | |
| QUOTE_IDENTIFIER | | TDV, Vertica | |
| QUOTE_LITERAL | | TDV, Vertica | |
| RADIANS | Yes | TDV, DataDirect Mainframe, DB2 DataDirect Mainframe, DB2, DB2 Mainframe, File, Greenplum, HSQLDB, Microsoft SQL Server, MySQL, Neoview, Netezza, PostgreSQL, Sybase, Sybase IQ, Vertica, XML File | |
| RAND | | SAP HANA | |
| RANDOM | | DB2, Hive, HSQLDB, Impala, Microsoft SQL Server, MySQL, Netezza, Oracle, PostgreSQL | |
| RANK | | TDV, DB2, Greenplum, Microsoft SQL Server, Netezza, Oracle, PostgreSQL, SAP HANA, Sybase IQ, Teradata, Vertica, Vertica OLAP | |
| RATIO_TO_REPORT | | TDV, Oracle | |
| REGEXP | | TDV, Hive, HSQLDB, Impala | |

| Function Name | TDV Repository | Supporting Data Source | Push - only |
|--------------------|----------------|---|-------------|
| REGEXP_EXT RACT | | TDV, Hive, HSQLDB, Impala | |
| REGEXP_REPLACEMENT | | TDV, Hive, HSQLDB, Impala | |
| REGR_AVGX | | TDV, Oracle, Vertica | |
| REGR_AVGY | | TDV, Oracle, Vertica | |
| REGR_COUNT | | TDV, Oracle, Vertica | |
| REGR_INTERCEPT | | TDV, Oracle, Vertica | |
| REGR_R2 | | TDV, Oracle, Vertica | |
| REGR_SLOPE | | TDV, Oracle, Vertica | |
| REGR_SXX | | TDV, Oracle, Vertica | |
| REGR_SXY | | TDV, Oracle, Vertica | |
| REGR_SYY | | TDV, Oracle, Vertica | |
| REPEAT | | Netezza, Vertica | |
| REPLACE | Yes | TDV, DataDirect Mainframe, DB2 DataDirect Mainframe, DB2, DB2 Mainframe, File, Greenplum, Hive, HSQLDB, Impala, Informix, Microsoft SQL Server, MySQL, Neoview, Oracle, PostgreSQL, SAP HANA, Teradata v15, Vertica, XML File | |
| REPLACE_REGEXPR | | SAP HANA | |
| REVERSE | | TDV, Hive, HSQLDB, Impala, Teradata v15 | |
| RIGHT | | SAP HANA, Teradata v15, Vertica | |
| RLIKE | | TDV, Hive, HSQLDB, Impala | |

| Function Name | TDV Repository | Supporting Data Source | Push - only |
|---------------|----------------|--|-------------|
| ROUND | Yes | TDV, DataDirect Mainframe, DB2 DataDirect Mainframe, DB2, DB2 Mainframe, File, Greenplum, HBASE, Hive, HSQLDB, Impala, Informix, Microsoft Access, Microsoft SQL Server, MySQL, Neoview, Netezza, Oracle, PostgreSQL, SAP HANA, Sybase, Sybase IQ, Teradata v15, Vertica, XML File | |
| ROW_NUMBER | | TDV, DB2, Greenplum, Microsoft SQL Server, Netezza, Oracle, PostgreSQL, SAP HANA, Sybase IQ, Teradata, Vertica, Vertica OLAP | |
| RPAD | | Hive, HSQLDB, Impala, Netezza, Oracle, PostgreSQL, SAP HANA, Teradata v15, Vertica | |
| RTRIM | Yes | TDV, DataDirect Mainframe, DB2 DataDirect Mainframe, DB2, DB2 Mainframe, File, Greenplum, HBASE, Hive, HSQLDB, Impala, Informix, Microsoft Access, Microsoft SQL Server, MySQL, Neoview, Netezza, Oracle, PostgreSQL, SAP HANA, Sybase, Sybase IQ, Teradata, Vertica, XML File | |
| SCORE_MP | | Netezza | |
| SEC_MP | | Netezza | |
| SECOND | | Impala, Microsoft SQL Server, SAP HANA, Vertica | |
| SIGN | | Netezza, Oracle, PostgreSQL, SAP HANA, Teradata v15 | |
| SIN | Yes | TDV, DataDirect Mainframe, DB2 DataDirect Mainframe, DB2, DB2 Mainframe, File, Greenplum, Hive, HSQLDB, Impala, Informix, Microsoft Access, Microsoft SQL Server, MySQL, Neoview, Netezza, Oracle, PostgreSQL, SAP HANA, Sybase, Sybase IQ, Teradata, Vertica, XML File | |
| SINH | | TDV | |
| SOUNDEX | | Oracle, Microsoft SQL Server, Hive, Netezza | |
| SPACE | Yes | DataDirect Mainframe, DB2 DataDirect Mainframe, DB2, DB2 Mainframe, Greenplum, Hive, HSQLDB, Impala, Microsoft Access, Microsoft SQL Server, MySQL, Netezza, Oracle, Vertica | |

| Function Name | TDV Repository | Supporting Data Source | Push - only |
|--------------------------|----------------|---|-------------|
| SPLIT_PART | | Vertica | Yes |
| SQRT | Yes | TDV, DataDirect Mainframe, DB2 DataDirect Mainframe, DB2, DB2 Mainframe, File, Greenplum, Hive, HSQLDB, Impala, Informix, Microsoft SQL Server, MySQL, Neoview, Netezza, Oracle, PostgreSQL, SAP HANA, Sybase, Sybase IQ, Teradata, Vertica, XML File | |
| STATEMENT_TIMESTAMP | | Vertica | Yes |
| STDDEV | | TDV, DB2, Greenplum, Hive, HSQLDB, Impala, Microsoft Access, Microsoft Excel, Microsoft SQL Server, MySQL, Netezza, Oracle, PostgreSQL, SAP HANA, Sybase IQ, Teradata, Vertica, Vertica OLAP | |
| STDDEV (analytical) | | TDV, Greenplum, Microsoft SQL Server, Netezza 4.5, Oracle, SAP HANA, Vertica | |
| STDDEV_POP | | TDV, DB2, Greenplum, HBASE, Hive, HSQLDB, Impala, Microsoft Access, Microsoft Excel, Microsoft SQL Server, MySQL, Netezza, Oracle, PostgreSQL, Sybase IQ, Teradata, Vertica | |
| STDDEV_POP (analytical) | | TDV, Greenplum, Netezza 4.5, Oracle, Teradata, Vertica, Vertica OLAP | |
| STDDEV_SAMP | | TDV, Greenplum, HBASE, Hive, HSQLDB, Impala, Microsoft Access, Microsoft Excel, Microsoft SQL Server, Netezza, Oracle, PostgreSQL, Sybase IQ, Teradata, Vertica | |
| STDDEV_SAMP (analytical) | | TDV, Greenplum, Netezza 4.5, Oracle, Teradata, Vertica, Vertica OLAP | |
| STRPOS | | Netezza, Oracle, PostgreSQL, Vertica | |
| SUBSTR | Yes | HBASE, Hive, HSQLDB, Impala, Netezza, Oracle, PostgreSQL, SAP HANA, Vertica | |
| SUBSTR_REG_EXPR | | SAP HANA | |

| Function Name | TDV Repository | Supporting Data Source | Push - only |
|------------------|----------------|--|-------------|
| SUBSTRING | Yes | TDV, DataDirect Mainframe, DB2 DataDirect Mainframe, DB2, DB2 Mainframe, File, Greenplum, Hive, HSQLDB, Impala, Informix, Microsoft SQL Server, MySQL, Neoview, Netezza, Oracle, PostgreSQL, SAP HANA, Sybase, Sybase IQ, Teradata, Vertica, XML File | |
| SUM | Yes | TDV, DataDirect Mainframe, DB2 DataDirect Mainframe, DB2, DB2 Mainframe, File, Greenplum, HBASE, Hive, HSQLDB, Impala, Informix, JDBC, Microsoft Access, Microsoft SQL Server, MySQL, Neoview, Netezza, Oracle, PostgreSQL, SAP HANA, Sybase, Sybase IQ, Teradata, Vertica, Vertica OLAP, XML File | |
| SUM (analytical) | | TDV, DB2, Microsoft SQL Server, Oracle, SAP HANA, Teradata, Vertica | |
| SUM_FLOAT | | TDV, Vertica | |
| SYSDATE | | TDV, Vertica | |
| TAN | Yes | TDV, DataDirect Mainframe, DB2 DataDirect Mainframe, DB2, DB2 Mainframe, File, Greenplum, Hive, HSQLDB, Impala, Informix, Microsoft Access, Microsoft SQL Server, MySQL, Neoview, Netezza, Oracle, PostgreSQL, SAP HANA, Sybase, Sybase IQ, Teradata, Vertica, XML File | |
| TANH | | TDV | |
| TEST | | Hive, HSQLDB, Impala | Yes |
| TIME_SLICE | | Vertica time series | |
| TIMEOFDAY | | Hive, HSQLDB, Impala, Netezza, Oracle, PostgreSQL, Vertica | |
| TIMESERIES | | Vertica time series | Yes |
| TIMESTAMP | | Netezza, Vertica | |
| TIMESTAMP_ROUND | | Vertica | Yes |

| Function Name | TDV Repository | Supporting Data Source | Push - only |
|-----------------|----------------|--|-------------|
| TIMESTAMP_TRUNC | | Vertica | Yes |
| TIMESTAMPADD | | Vertica | Yes |
| TIMESTAMPDIFF | | Vertica | Yes |
| TO_BITSTRING | | Vertica | Yes |
| TO_CANONICAL | | | |
| TO_CHAR | Yes | TDV, DataDirect Mainframe, DB2 DataDirect Mainframe, DB2, DB2 Mainframe, File, Greenplum, HBASE, Hive, HSQLDB, Impala, Informix, Microsoft Access, Microsoft SQL Server, MySQL, Neoview, Netezza, Oracle, PostgreSQL, Sybase, Sybase IQ, Teradata v15, Vertica, XML File | |
| TO_DATE | | TDV, DataDirect Mainframe, DB2 DataDirect Mainframe, DB2, DB2 Mainframe, File, Greenplum, Hive, HSQLDB, Impala, Informix, Microsoft Access, Microsoft SQL Server, Neoview, Netezza, Oracle, PostgreSQL, SAP HANA, Sybase, Sybase IQ, Teradata, Vertica, XML File | |
| TO_FIRST_VALUE | | Vertica time series | |
| TO_HEX | | Vertica | Yes |
| TO_LAST_VALUE | | Vertica time series | |
| TO_NUMBER | Yes | TDV, DataDirect Mainframe, DB2 DataDirect Mainframe, DB2, DB2 Mainframe, File, Greenplum, Informix, Microsoft Access, Microsoft SQL Server, MySQL, Neoview, Netezza, Oracle, PostgreSQL, Sybase, Sybase IQ, Vertica, XML File | |

| Function Name | TDV Repository | Supporting Data Source | Push - only |
|-----------------------|----------------|---|-------------|
| TO_TIMESTAMP | | TDV, DataDirect Mainframe, DB2 DataDirect Mainframe, DB2, DB2 Mainframe, File, Greenplum, Hive, HSQLDB, Impala, Microsoft Access, Microsoft SQL Server, Neoview, Netezza, Oracle, PostgreSQL, SAP HANA, Sybase, Sybase IQ, Teradata v15, Vertica, XML File | |
| TO_TIMESTAMP_TZ | | Teradata v15 | |
| TO_VARCHAR | | SAP HANA | |
| TRANSACTION_TIMESTAMP | | Vertica | Yes |
| TRANSLATE | | Netezza, Oracle, PostgreSQL, Teradata v15, Vertica | |
| TRIM | Yes | TDV, DataDirect Mainframe, DB2 DataDirect Mainframe, DB2, DB2 Mainframe, File, Greenplum, Hive, HSQLDB, Impala, Informix, Microsoft Access, Microsoft SQL Server, MySQL, Neoview, Netezza, Oracle, PostgreSQL, SAP HANA, Sybase, Sybase IQ, Teradata, Vertica, XML File | |
| TRIMBOTH | | | |
| TRIMLEADING | | | |
| TRIMTRAILING | | | |
| TRUNC | | Netezza, Oracle, PostgreSQL, Teradata v15, Vertica | |
| TRUNCATE | | | |
| TS_FIRST_VALUE | | Vertica time series | Yes |
| TS_LAST_VALUE | | Vertica time series | Yes |
| TZ_OFFSET | | | |

| Function Name | TDV Repository | Supporting Data Source | Push - only |
|----------------------|----------------|--|-------------|
| TZCONVERTOR | | MySQL | |
| UCASE | | HBASE, HSQLDB, Impala, SAP HANA | |
| UNICHR | | Netezza, Oracle | |
| UNICODE | | Netezza, SAP HANA | |
| UNIX_TIMESTAMP | | Hive, HSQLDB, Impala | |
| UPPER | Yes | TDV, DataDirect Mainframe, DB2 DataDirect Mainframe, DB2, DB2 Mainframe, File, Greenplum, HBASE, Hive, HSQLDB, Impala, Informix, Microsoft Access, Microsoft SQL Server, MySQL, Neoview, Netezza, Oracle, PostgreSQL, SAP HANA, Sybase, Sybase IQ, Teradata, Vertica, XML File | |
| UTC_TO_TIMEESTAMP | | Hive, HSQLDB, Impala, MySQL | |
| V6_ATON | | Vertica | Yes |
| V6_NTOA | | Vertica | Yes |
| V6_SUBNETA | | Vertica | Yes |
| V6_SUBNETN | | Vertica | Yes |
| V6_TYPE | | Vertica | Yes |
| VAR (analytical) | | SAP HANA | |
| VAR_POP | | TDV, Greenplum, Hive, HSQLDB, Impala, Microsoft SQL Server, Netezza, Oracle, PostgreSQL, Sybase IQ, Teradata, Vertica | |
| VAR_POP (analytical) | | TDV, Greenplum, Netezza 4.5, Oracle, Teradata, Vertica, Vertica OLAP | |

| Function Name | TDV Repository | Supporting Data Source | Push - only |
|-----------------------|----------------|---|-------------|
| VAR_SAMP | | TDV, Greenplum, Hive, HSQLDB, Impala, Netezza, Oracle, PostgreSQL, Sybase IQ, Teradata, Vertica | |
| VAR_SAMP (analytical) | | TDV, Greenplum, Netezza 4.5, Oracle, Teradata, Vertica, Vertica OLAP | |
| VARIANCE | | TDV, DB2, Greenplum, Hive, HSQLDB, Impala, Microsoft SQL Server, Netezza, Oracle, PostgreSQL, SAP HANA, Sybase IQ, Vertica | |
| VARIANCE (analytical) | | TDV, Greenplum, Microsoft SQL Server, Netezza 4.5, Oracle, Vertica, Vertica OLAP | |
| VARIANCE_POP | | TDV, DB2, Greenplum, Microsoft Access, Microsoft Excel, Microsoft SQL Server, MySQL, Netezza, Oracle, PostgreSQL, Teradata, Vertica | |
| VARIANCE_SAMP | | TDV, Greenplum, Microsoft Access, Microsoft Excel, Microsoft SQL Server, Netezza, Oracle, PostgreSQL, Teradata, Vertica | |
| WEEK | | Microsoft SQL Server, SAP HANA, Vertica | |
| WEEK_ISO | | Vertica | Yes |
| WEEKDAY | | TDV, SAP HANA | |
| XMLAGG | | Oracle | |
| XMLATTRIBUTES | | DataDirect Mainframe, DB2 DataDirect Mainframe (with XML Extender), Oracle | |
| XMLCOMMENT | | DataDirect Mainframe, DB2 DataDirect Mainframe (with XML Extender) | |
| XMLCONCAT | | DataDirect Mainframe, DB2 DataDirect Mainframe (with XML Extender), Oracle | |
| XMLDOCUMENT | | DataDirect Mainframe, DB2 DataDirect Mainframe (with XML Extender) | |
| XMLEMENT | | DataDirect Mainframe, DB2 DataDirect Mainframe (with XML Extender), Oracle | |

| Function Name | TDV Repository | Supporting Data Source | Push - only |
|---------------|----------------|---|-------------|
| XMLFOREST | | DataDirect Mainframe, DB2 DataDirect Mainframe (with XML Extender), Oracle | |
| XMLNAMESPACES | | DataDirect Mainframe, DB2 DataDirect Mainframe (with XML Extender) | |
| XMLPI | | DataDirect Mainframe, DB2 DataDirect Mainframe (with XML Extender) | |
| XMLQUERY | | DataDirect Mainframe, DB2, DB2 DataDirect Mainframe (with XML Extender) | |
| XMLTEXT | | DataDirect Mainframe, DB2 DataDirect Mainframe (with XML Extender) | |
| XPATH | | | |
| XSLT | | | |
| YEAR | Yes | TDV, DataDirect Mainframe, DB2 DataDirect Mainframe, DB2, DB2 Mainframe, File, Greenplum, Hive, HSQLDB, Impala, Informix, Microsoft Access, Microsoft SQL Server, MySQL, Neoview, Netezza, Oracle, SAP HANA, Sybase, Sybase IQ, Teradata, Vertica, XML File | |
| YEAR_ISO | | TDV, Vertica | |

Time Zones

This topic describes the time zone designations that can be used in the TDV implementation of the TZCONVERTOR function.

- Java has deprecated three-letter acronyms for time zones. Despite this, Java still supports a few of them, such as UTC, GMT, and EST. If you intend to use any of them in production environment, thoroughly test them first, because using them can lead to incompatibilities or errors.
- Time zone information varies by locale, platform, and operating system version. Therefore the list in the table below is not definitive.
- Be aware that a timestamp in a locale that supports daylight saving time may or may not convert to a value one hour later (equivalent to an unaltered time zone to the east of it).
- The TDV implementation of TZCONVERTOR does not support offset notation such as GMT+5.

| | | |
|----------------------|---------------------|--------------------|
| Africa/Abidjan | Africa/Accra | Africa/Addis_Ababa |
| Africa/Algiers | Africa/Asmara | Africa/Asmera |
| Africa/Bamako | Africa/Bangui | Africa/Banjul |
| Africa/Bissau | Africa/Blantyre | Africa/Brazzaville |
| Africa/Bujumbura | Africa/Cairo | Africa/Casablanca |
| Africa/Ceuta | Africa/Conakry | Africa/Dakar |
| Africa/Dar_es_Salaam | Africa/Djibouti | Africa/Douala |
| Africa/El_Aaiun | Africa/Freetown | Africa/Gaborone |
| Africa/Harare | Africa/Johannesburg | Africa/Juba |
| Africa/Kampala | Africa/Khartoum | Africa/Kigali |
| Africa/Kinshasa | Africa/Lagos | Africa/Libreville |
| Africa/Lome | Africa/Luanda | Africa/Lubumbashi |
| Africa/Lusaka | Africa/Malabo | Africa/Maputo |

| | | |
|--------------------------------|----------------------------------|--------------------------------|
| Africa/Maseru | Africa/Mbabane | Africa/Mogadishu |
| Africa/Monrovia | Africa/Nairobi | Africa/Ndjamena |
| Africa/Niamey | Africa/Nouakchott | Africa/Ouagadougou |
| Africa/Porto-Novo | Africa/Sao_Tome | Africa/Timbuktu |
| Africa/Tripoli | Africa/Tunis | Africa/Windhoek |
| America/Adak | America/Anchorage | America/Anguilla |
| America/Antigua | America/Araguaina | America/Argentina/Buenos_Aires |
| America/Argentina/Catamarca | America/Argentina/ComodRivadavia | America/Argentina/Cordoba |
| America/Argentina/Jujuy | America/Argentina/La_Rioja | America/Argentina/Mendoza |
| America/Argentina/Rio_Gallegos | America/Argentina/Salta | America/Argentina/San_Juan |
| America/Argentina/San_Luis | America/Argentina/Tucuman | America/Argentina/Ushuaia |
| America/Aruba | America/Asuncion | America/Atikokan |
| America/Atka | America/Bahia | America/Bahia_Banderas |
| America/Barbados | America/Belem | America/Belize |
| America/Blanc-Sablon | America/Boa_Vista | America/Bogota |
| America/Boise | America/Buenos_Aires | America/Cambridge_Bay |
| America/Campo_Grande | America/Cancun | America/Caracas |
| America/Catamarca | America/Cayenne | America/Cayman |
| America/Chicago | America/Chihuahua | America/Coral_Harbour |
| America/Cordoba | America/Costa_Rica | America/Creston |
| America/Cuiaba | America/Curacao | America/Danmarkshavn |

| | | |
|---------------------------|-----------------------------|------------------------------|
| America/Dawson | America/Dawson_Creek | America/Denver |
| America/Detroit | America/Dominica | America/Edmonton |
| America/Eirunepe | America/El_Salvador | America/Ensenada |
| America/Fort_Wayne | America/Fortaleza | America/Glace_Bay |
| America/Godthab | America/Goose_Bay | America/Grand_Turk |
| America/Grenada | America/Guadeloupe | America/Guatemala |
| America/Guayaquil | America/Guyana | America/Halifax |
| America/Havana | America/Hermosillo | America/Indiana/Indianapolis |
| America/Indiana/Knox | America/Indiana/Marengo | America/Indiana/Petersburg |
| America/Indiana/Tell_City | America/Indiana/Vevay | America/Indiana/Vincennes |
| America/Indiana/Winamac | America/Indianapolis | America/Inuvik |
| America/Iqaluit | America/Jamaica | America/Jujuy |
| America/Juneau | America/Kentucky/Louisville | America/Kentucky/Monticello |
| America/Knox_IN | America/Kralendijk | America/La_Paz |
| America/Lima | America/Los_Angeles | America/Louisville |
| America/Lower_Princes | America/Maceio | America/Managua |
| America/Manaus | America/Marigot | America/Martinique |
| America/Matamoros | America/Mazatlan | America/Mendoza |
| America/Menominee | America/Merida | America/Metlakatla |
| America/Mexico_City | America/Miquelon | America/Moncton |
| America/Monterrey | America/Montevideo | America/Montreal |
| America/Montserrat | America/Nassau | America/New_York |
| America/Nipigon | America/Nome | America/Noronha |

| | | |
|-----------------------------|-----------------------------|--------------------------------|
| America/North_Dakota/Beulah | America/North_Dakota/Center | America/North_Dakota/New_Salem |
| America/Ojinaga | America/Panama | America/Pangnirtung |
| America/Paramaribo | America/Phoenix | America/Port-au-Prince |
| America/Port_of_Spain | America/Porto_Acre | America/Porto_Velho |
| America/Puerto_Rico | America/Rainy_River | America/Rankin_Inlet |
| America/Recife | America/Regina | America/Resolute |
| America/Rio_Branco | America/Rosario | America/Santa_Isabel |
| America/Santarem | America/Santiago | America/Santo_Domingo |
| America/Sao_Paulo | America/Scoresbysund | America/Shiprock |
| America/Sitka | America/St_Barthelemy | America/St_Johns |
| America/St_Kitts | America/St_Lucia | America/St_Thomas |
| America/St_Vincent | America/Swift_Current | America/Tegucigalpa |
| America/Thule | America/Thunder_Bay | America/Tijuana |
| America/Toronto | America/Tortola | America/Vancouver |
| America/Virgin | America/Whitehorse | America/Winnipeg |
| America/Yakutat | America/Yellowknife | Antarctica/Casey |
| Antarctica/Davis | Antarctica/DumontDURville | Antarctica/Macquarie |
| Antarctica/Mawson | Antarctica/McMurdo | Antarctica/Palmer |
| Antarctica/Rothera | Antarctica/South_Pole | Antarctica/Syowa |
| Antarctica/Vostok | Arctic/Longyearbyen | Asia/Aden |
| Asia/Almaty | Asia/Amman | Asia/Anadyr |
| Asia/Aqtau | Asia/Aqtobe | Asia/Ashgabat |
| Asia/Ashkhabad | Asia/Baghdad | Asia/Bahrain |
| Asia/Baku | Asia/Bangkok | Asia/Beijing |

| | | |
|-------------------|-------------------|--------------------|
| Asia/Beirut | Asia/Bishkek | Asia/Brunei |
| Asia/Calcutta | Asia/Choibalsan | Asia/Chongqing |
| Asia/Chungking | Asia/Colombo | Asia/Dacca |
| Asia/Damascus | Asia/Dhaka | Asia/Dili |
| Asia/Dubai | Asia/Dushanbe | Asia/Gaza |
| Asia/Harbin | Asia/Hebron | Asia/Ho_Chi_Minh |
| Asia/Hong_Kong | Asia/Hovd | Asia/Irkutsk |
| Asia/Istanbul | Asia/Jakarta | Asia/Jayapura |
| Asia/Jerusalem | Asia/Kabul | Asia/Kamchatka |
| Asia/Karachi | Asia/Kashgar | Asia/Kathmandu |
| Asia/Katmandu | Asia/Kolkata | Asia/Krasnoyarsk |
| Asia/Kuala_Lumpur | Asia/Kuching | Asia/Kuwait |
| Asia/Macao | Asia/Macau | Asia/Magadan |
| Asia/Makassar | Asia/Manila | Asia/Muscat |
| Asia/Nicosia | Asia/Novokuznetsk | Asia/Novosibirsk |
| Asia/Omsk | Asia/Oral | Asia/Phnom_Penh |
| Asia/Pontianak | Asia/Pyongyang | Asia/Qatar |
| Asia/Qyzylorda | Asia/Rangoon | Asia/Riyadh |
| Asia/Riyadh87 | Asia/Riyadh88 | Asia/Riyadh89 |
| Asia/Saigon | Asia/Sakhalin | Asia/Samarkand |
| Asia/Seoul | Asia/Shanghai | Asia/Singapore |
| Asia/Taipei | Asia/Tashkent | Asia/Tbilisi |
| Asia/Tehran | Asia/Tel_Aviv | Asia/Thimbu |
| Asia/Thimphu | Asia/Tokyo | Asia/Ujung_Pandang |

| | | |
|--------------------------|---------------------|------------------------|
| Asia/Ulaanbaatar | Asia/Ulan_Bator | Asia/Urumqi |
| Asia/Vientiane | Asia/Vladivostok | Asia/Yakutsk |
| Asia/Yekaterinburg | Asia/Yerevan | Atlantic/Azores |
| Atlantic/Bermuda | Atlantic/Canary | Atlantic/Cape_Verde |
| Atlantic/Faeroe | Atlantic/Faroe | Atlantic/Jan_Mayen |
| Atlantic/Madeira | Atlantic/Reykjavik | Atlantic/South_Georgia |
| Atlantic/St_Helena | Atlantic/Stanley | Australia/ACT |
| Australia/Adelaide | Australia/Brisbane | Australia/Broken_Hill |
| Australia/Canberra | Australia/Currie | Australia/Darwin |
| Australia/Eucla | Australia/Hobart | Australia/LHI |
| Australia/Lindeman | Australia/Lord_Howe | Australia/Melbourne |
| Australia/NSW | Australia/North | Australia/Perth |
| Australia/Queensland | Australia/South | Australia/Sydney |
| Australia/Tasmania | Australia/Victoria | Australia/West |
| Australia/Yancowinna | Brazil/Acre | Brazil/DeNoronha |
| Brazil/East | Brazil/West | CET |
| CST6CDT | Canada/Atlantic | Canada/Central |
| Canada/East-Saskatchewan | Canada/Eastern | Canada/Mountain |
| Canada/Newfoundland | Canada/Pacific | Canada/Saskatchewan |
| Canada/Yukon | Chile/Continental | Chile/EasterIsland |
| Cuba | EET | EST5EDT |
| Egypt | Eire | Etc/GMT |
| Etc/GMT+0 | Etc/GMT+1 | Etc/GMT+10 |
| Etc/GMT+11 | Etc/GMT+12 | Etc/GMT+2 |

| | | |
|-------------------|-------------------|--------------------|
| Etc/GMT+3 | Etc/GMT+4 | Etc/GMT+5 |
| Etc/GMT+6 | Etc/GMT+7 | Etc/GMT+8 |
| Etc/GMT+9 | Etc/GMT-0 | Etc/GMT-1 |
| Etc/GMT-10 | Etc/GMT-11 | Etc/GMT-12 |
| Etc/GMT-13 | Etc/GMT-14 | Etc/GMT-2 |
| Etc/GMT-3 | Etc/GMT-4 | Etc/GMT-5 |
| Etc/GMT-6 | Etc/GMT-7 | Etc/GMT-8 |
| Etc/GMT-9 | Etc/GMT0 | Etc/Greenwich |
| Etc/UCT | Etc/UTC | Etc/Universal |
| Etc/Zulu | Europe/Amsterdam | Europe/Andorra |
| Europe/Athens | Europe/Belfast | Europe/Belgrade |
| Europe/Berlin | Europe/Bratislava | Europe/Brussels |
| Europe/Bucharest | Europe/Budapest | Europe/Chisinau |
| Europe/Copenhagen | Europe/Dublin | Europe/Gibraltar |
| Europe/Guernsey | Europe/Helsinki | Europe/Isle_of_Man |
| Europe/Istanbul | Europe/Jersey | Europe/Kaliningrad |
| Europe/Kiev | Europe/Lisbon | Europe/Ljubljana |
| Europe/London | Europe/Luxembourg | Europe/Madrid |
| Europe/Malta | Europe/Mariehamn | Europe/Minsk |
| Europe/Monaco | Europe/Moscow | Europe/Nicosia |
| Europe/Oslo | Europe/Paris | Europe/Podgorica |
| Europe/Prague | Europe/Riga | Europe/Rome |
| Europe/Samara | Europe/San_Marino | Europe/Sarajevo |
| Europe/Simferopol | Europe/Skopje | Europe/Sofia |

| | | |
|---------------------|-----------------|------------------|
| Europe/Stockholm | Europe/Tallinn | Europe/Tirane |
| Europe/Tiraspol | Europe/Uzhgorod | Europe/Vaduz |
| Europe/Vatican | Europe/Vienna | Europe/Vilnius |
| Europe/Volgograd | Europe/Warsaw | Europe/Zagreb |
| Europe/Zaporozhye | Europe/Zurich | Factory |
| GB | GB-Eire | GMT |
| GMT+0 | GMT+1 | GMT+10 |
| GMT+11 | GMT+12 | GMT+13 |
| GMT+14 | GMT+2 | GMT+3 |
| GMT+4 | GMT+5 | GMT+6 |
| GMT+7 | GMT+8 | GMT+9 |
| GMT-0 | GMT-1 | GMT-10 |
| GMT-11 | GMT-12 | GMT-2 |
| GMT-3 | GMT-4 | GMT-5 |
| GMT-6 | GMT-7 | GMT-8 |
| GMT-9 | GMT0 | Greenwich |
| HST | Hongkong | Iceland |
| Indian/Antananarivo | Indian/Chagos | Indian/Christmas |
| Indian/Cocos | Indian/Comoro | Indian/Kerguelen |
| Indian/Mahe | Indian/Maldives | Indian/Mauritius |
| Indian/Mayotte | Indian/Reunion | Iran |
| Israel | Jamaica | Japan |
| Kwajalein | Libya | MET |
| MST | MST7MDT | Mexico/BajaNorte |

| | | |
|-------------------|---------------------|----------------------|
| Mexico/BajaSur | Mexico/General | Mideast/Riyadh87 |
| Mideast/Riyadh88 | Mideast/Riyadh89 | NZ |
| NZ-CHAT | Navajo | PRC |
| PST8PDT | Pacific/Apia | Pacific/Auckland |
| Pacific/Chatham | Pacific/Chuuk | Pacific/Easter |
| Pacific/Efate | Pacific/Enderbury | Pacific/Fakaofo |
| Pacific/Fiji | Pacific/Funafuti | Pacific/Galapagos |
| Pacific/Gambier | Pacific/Guadalcanal | Pacific/Guam |
| Pacific/Honolulu | Pacific/Johnston | Pacific/Kiritimati |
| Pacific/Kosrae | Pacific/Kwajalein | Pacific/Majuro |
| Pacific/Marquesas | Pacific/Midway | Pacific/Nauru |
| Pacific/Niue | Pacific/Norfolk | Pacific/Noumea |
| Pacific/Pago_Pago | Pacific/Palau | Pacific/Pitcairn |
| Pacific/Pohnpei | Pacific/Ponape | Pacific/Port_Moresby |
| Pacific/Rarotonga | Pacific/Saipan | Pacific/Samoa |
| Pacific/Tahiti | Pacific/Tarawa | Pacific/Tongatapu |
| Pacific/Truk | Pacific/Wake | Pacific/Wallis |
| Pacific/Yap | Poland | Portugal |
| ROC | ROK | Singapore |
| Turkey | UCT | US/Alaska |
| US/Aleutian | US/Arizona | US/Central |
| US/East-Indiana | US/Eastern | US/Hawaii |
| US/Indiana-Starke | US/Michigan | US/Mountain |
| US/Pacific | US/Pacific-New | US/Samoa |

| | | |
|-----|-----------|------|
| UTC | Universal | W-SU |
| WET | Zulu | |