Trafodion SQL Reference Manual

Part Number: T775-110-001
Published: April 2015
Edition: Trafodion Release 1.1.0

Legal Notice

Acknowledgements

Contents

About This Document....17
Intended Audience....17
New and Changed Information in This Edition....17
Document Organization....17
Notation Conventions....18
General Syntax Notation....18
Publishing History....20
We Encourage Your Comments....20
1 Introduction....21
SQL Language....21
Using Trafodion SQL to Access HBase Tables....21
Initializing the Trafodion Metadata....21
Ways to Access HBase Tables....22
Trafodion SQL Tables Versus Native HBase Tables....23
Supported SQL Statements With HBase Tables....23
Using Trafodion SQL to Access Hive Tables....24
ANSI Names for Hive Tables....24
Type Mapping From Hive to Trafodion SQL....24
Supported SQL Statements With Hive Tables....24
Data Consistency and Access Options....25
READ COMMITTED....25
Transaction Management....25
User-Defined and System-Defined Transactions....26
Rules for DML Statements....26
Effect of AUTOCOMMIT Option....26
Concurrency....26
Transaction Isolation Levels....26
ANSI Compliance and Trafodion SQL Extensions....27
ANSI-Compliant Statements....27
Statements That Are Trafodion SQL Extensions....28
ANSI-Compliant Functions....28
Trafodion SQL Error Messages....29
2 SQL Statements....30
Categories....30
Data Definition Language (DDL) Statements....30
Data Manipulation Language (DML) Statements....31
Transaction Control Statements....31
Data Control and Security Statements....31
Stored Procedure and User-Defined Function Statements....32
Prepared Statements....32
Control Statements....33
Object Naming Statements....33
SHOW, GET, and EXPLAIN Statements....33
ALTER LIBRARY Statement....34
Syntax Description of ALTER LIBRARY....34
Considerations for ALTER LIBRARY....34
Examples of ALTER LIBRARY....34
ALTER TABLE Statement....36
Syntax Description of ALTER TABLE....37
Considerations for ALTER TABLE....40

Example of ALTER TABLE....40
ALTER USER Statement....41
Syntax Description of ALTER USER....41
Considerations for ALTER USER....41
Examples of ALTER USER....41
BEGIN WORK Statement....42
Considerations for BEGIN WORK....42
Example of BEGIN WORK....42
CALL Statement....43
Syntax Description of CALL....43
Considerations for CALL....43
Examples of CALL....44
COMMIT WORK Statement....46
Considerations for COMMIT WORK....46
Example of COMMIT WORK....46
CONTROL QUERY CANCEL Statement....47
Syntax Description of CONTROL QUERY CANCEL....47
Considerations for CONTROL QUERY CANCEL....47
Example of CONTROL QUERY CANCEL....48
CONTROL QUERY DEFAULT Statement....49
Syntax Description of CONTROL QUERY DEFAULT....49
Considerations for CONTROL QUERY DEFAULT....49
Examples of CONTROL QUERY DEFAULT....49
CREATE FUNCTION Statement....50
Syntax Description of CREATE FUNCTION....50
Considerations for CREATE FUNCTION....52
Examples of CREATE FUNCTION....52
CREATE INDEX Statement....53
Syntax Description of CREATE INDEX....53
Considerations for CREATE INDEX....54
Examples of CREATE INDEX....55
CREATE LIBRARY Statement....56
Syntax Description of CREATE LIBRARY....56
Considerations for CREATE LIBRARY....56
Examples of CREATE LIBRARY....57
CREATE PROCEDURE Statement....58
Syntax Description of CREATE PROCEDURE....58
Considerations for CREATE PROCEDURE....62
Examples of CREATE PROCEDURE....63
CREATE ROLE Statement....66
Syntax Description of CREATE ROLE....66
Considerations for CREATE ROLE....66
Examples of CREATE ROLE....66
CREATE SCHEMA Statement....67
Syntax Description of CREATE SCHEMA....67
Considerations for CREATE SCHEMA....67
Examples of CREATE SCHEMA....68
CREATE TABLE Statement....69
Syntax Description of CREATE TABLE....70
Considerations for CREATE TABLE....74
Authorization and Availability Requirements....74
Considerations for CREATE VOLATILE TABLE....75
Considerations for CREATE TABLE ... LIKE....77
Considerations for CREATE TABLE AS....78
Trafodion SQL Extensions to CREATE TABLE....78

Examples of CREATE TABLE....79
Examples of CREATE TABLE AS....79
CREATE VIEW Statement....81
Syntax Description of CREATE VIEW....81
Considerations for CREATE VIEW....82
Examples of CREATE VIEW....84
DELETE Statement....86
Syntax Description of DELETE....86
Considerations for DELETE....86
Examples of DELETE....87
DROP FUNCTION Statement....88
Syntax Description of DROP FUNCTION....88
Considerations for DROP FUNCTION....88
Examples of DROP FUNCTION....88
DROP INDEX Statement....89
Syntax Description of DROP INDEX....89
Considerations for DROP INDEX....89
Examples of DROP INDEX....89
DROP LIBRARY Statement....90
Syntax Description of DROP LIBRARY....90
Considerations for DROP LIBRARY....90
Examples of DROP LIBRARY....90
DROP PROCEDURE Statement....92
Syntax Description of DROP PROCEDURE....92
Considerations for DROP PROCEDURE....92
Examples of DROP PROCEDURE....92
DROP ROLE Statement....93
Syntax Description of DROP ROLE....93
Considerations for DROP ROLE....93
Before You Drop a Role....93
Active Sessions for the User....93
Examples of DROP ROLE....93
DROP SCHEMA Statement....95
Syntax Description of DROP SCHEMA....95
Considerations for DROP SCHEMA....95
Example of DROP SCHEMA....95
DROP TABLE Statement....96
Syntax Description of DROP TABLE....96
Considerations for DROP TABLE....96
Examples of DROP TABLE....96
DROP VIEW Statement....97
Syntax Description of DROP VIEW....97
Considerations for DROP VIEW....97
Example of DROP VIEW....97
EXECUTE Statement....98
Syntax Description of EXECUTE....98
Considerations for EXECUTE....99
Examples of EXECUTE....99
EXPLAIN Statement....101
Syntax Description of EXPLAIN....101
Considerations for EXPLAIN....101
GET Statement....103
Syntax Description of GET....103
Considerations for GET....104
Examples of GET....105

GET HBASE OBJECTS Statement....107
Syntax Description of GET HBASE OBJECTS....107
Examples of GET HBASE OBJECTS....107
GET VERSION OF METADATA Statement....109
Considerations for GET VERSION OF METADATA....109
Examples of GET VERSION OF METADATA....109
GET VERSION OF SOFTWARE Statement....110
Considerations for GET VERSION OF SOFTWARE....110
Examples of GET VERSION OF SOFTWARE....110
GRANT Statement....111
Syntax Description of GRANT....111
Considerations for GRANT....112
Examples of GRANT....112
GRANT COMPONENT PRIVILEGE Statement....114
Syntax Description of GRANT COMPONENT PRIVILEGE....114
Considerations for GRANT COMPONENT PRIVILEGE....116
Example of GRANT COMPONENT PRIVILEGE....116
GRANT ROLE Statement....117
Syntax Description of GRANT ROLE....117
Considerations for GRANT ROLE....117
Example of GRANT ROLE....117
INSERT Statement....118
Syntax Description of INSERT....118
Considerations for INSERT....118
Examples of INSERT....120
INVOKE Statement....122
Syntax Description of INVOKE....122
Considerations for INVOKE....122
Example of INVOKE....122
MERGE Statement....123
Syntax Description of MERGE....123
Considerations for MERGE....123
Example of MERGE....125
PREPARE Statement....126
Syntax Description of PREPARE....126
Considerations for PREPARE....126
Examples of PREPARE....126
REGISTER USER Statement....128
Syntax Description of REGISTER USER....128
Considerations for REGISTER USER....128
Examples of REGISTER USER....129
REVOKE Statement....130
Syntax Description of REVOKE....130
Considerations for REVOKE....131
Examples of REVOKE....131
REVOKE COMPONENT PRIVILEGE Statement....133
Syntax Description of REVOKE COMPONENT PRIVILEGE....133
Considerations for REVOKE COMPONENT PRIVILEGE....134
Example of REVOKE COMPONENT PRIVILEGE....134
REVOKE ROLE Statement....135
Syntax Description of REVOKE ROLE....135
Considerations for REVOKE ROLE....135
Examples of REVOKE ROLE....136
ROLLBACK WORK Statement....137
Syntax Description of ROLLBACK WORK....137

Considerations for ROLLBACK WORK....137
Example of ROLLBACK WORK....137
SELECT Statement....138
Syntax Description of SELECT....140
Considerations for SELECT....146
Considerations for Select List....148
Considerations for GROUP BY....148
Considerations for ORDER BY....148
Considerations for UNION....149
Examples of SELECT....150
SET SCHEMA Statement....156
Syntax Description of SET SCHEMA....156
Considerations for SET SCHEMA....156
Example of SET SCHEMA....156
SET TRANSACTION Statement....157
Syntax Description of SET TRANSACTION....157
Considerations for SET TRANSACTION....157
Examples of SET TRANSACTION....157
SHOWCONTROL Statement....159
Syntax Description of SHOWCONTROL....159
Example of SHOWCONTROL....159
SHOWDDL Statement....160
Syntax Description of SHOWDDL....160
Considerations for SHOWDDL....160
Examples of SHOWDDL....161
SHOWDDL SCHEMA Statement....163
Syntax Description for SHOWDDL SCHEMA....163
Considerations for SHOWDDL SCHEMA....163
Example of SHOWDDL SCHEMA....163
SHOWSTATS Statement....164
Syntax Description of SHOWSTATS....164
Considerations for SHOWSTATS....165
Examples of SHOWSTATS....165
TABLE Statement....167
Considerations for TABLE....167
Example of TABLE....167
UNREGISTER USER Statement....168
Syntax Description of UNREGISTER USER....168
Considerations for UNREGISTER USER....168
Example of UNREGISTER USER....168
UPDATE Statement....169
Syntax Description of UPDATE....169
Considerations for UPDATE....170
Examples of UPDATE....172
UPSERT Statement....173
Syntax Description of UPSERT....173
Examples of UPSERT....173
VALUES Statement....175
Considerations for VALUES....175
Examples of VALUES....175
3 SQL Utilities....176
LOAD Statement....177
Syntax Description of LOAD....177
Considerations for LOAD....178

Example of LOAD....179
POPULATE INDEX Utility....180
Syntax Description of POPULATE INDEX....180
Considerations for POPULATE INDEX....180
Examples of POPULATE INDEX....181
PURGEDATA Utility....182
Syntax Description of PURGEDATA....182
Considerations for PURGEDATA....182
Example of PURGEDATA....182
UNLOAD Statement....183
Syntax Description of UNLOAD....183
Considerations for UNLOAD....184
Example of UNLOAD....184
UPDATE STATISTICS Statement....186
Syntax Description of UPDATE STATISTICS....186
Considerations for UPDATE STATISTICS....189
Examples of UPDATE STATISTICS....191
4 SQL Language Elements....192
Authorization IDs....193
Character Sets....193
Columns....193
Column References....193
Derived Column Names....193
Column Default Settings....194
Constraints....195
Creating or Adding Constraints on SQL Tables....195
Constraint Names....195
Correlation Names....196
Explicit Correlation Names....196
Implicit Correlation Names....196
Examples of Correlation Names....196
Database Objects....197
Ownership....197
Database Object Names....198
Logical Names for SQL Objects....198
SQL Object Namespaces....198
Data Types....199
Comparable and Compatible Data Types....201
Character String Data Types....204
Datetime Data Types....205
Interval Data Types....207
Numeric Data Types....209
Expressions....211
Character Value Expressions....211
Datetime Value Expressions....212
Interval Value Expressions....215
Numeric Value Expressions....218
Identifiers....221
Regular Identifiers....221
Delimited Identifiers....221
Case-Insensitive Delimited Identifiers....221
Examples of Identifiers....221
Indexes....222
SQL Indexes....222

Keys....223
Clustering Keys....223
SYSKEY....223
Index Keys....223
Primary Keys....223
Literals....224
Character String Literals....224
Datetime Literals....226
Interval Literals....227
Numeric Literals....229
Null....231
Using Null Versus Default Values....231
Defining Columns That Allow or Prohibit Null....231
Predicates....233
BETWEEN Predicate....233
Comparison Predicates....234
EXISTS Predicate....238
IN Predicate....239
LIKE Predicate....241
NULL Predicate....243
Quantified Comparison Predicates....244
Privileges....247
Roles....248
Schemas....249
Creating and Dropping Schemas....249
Search Condition....250
Considerations for Search Condition....250
Examples of Search Condition....251
Subquery....252
SELECT Form of a Subquery....252
Using Subqueries to Provide Comparison Values....252
Nested Subqueries When Providing Comparison Values....252
Correlated Subqueries When Providing Comparison Values....252
Tables....254
Base Tables and Views....254
Example of a Base Table....254
Views....255
SQL Views....255
Example of a View....255
5 SQL Clauses....256
DEFAULT Clause....257
Examples of DEFAULT....257
FORMAT Clause....259
Considerations for Date Formats....260
Considerations for Other Formats....260
Examples of FORMAT....260
SAMPLE Clause....261
Considerations for SAMPLE....262
Examples of SAMPLE....262
SEQUENCE BY Clause....268
Considerations for SEQUENCE BY....268
Examples of SEQUENCE BY....269
TRANSPOSE Clause....271
Considerations for TRANSPOSE....272

Examples of TRANSPOSE....273
6 SQL Functions and Expressions....278
Categories....278
Standard Normalization....278
Aggregate (Set) Functions....278
Character String Functions....279
Datetime Functions....280
Mathematical Functions....281
Sequence Functions....282
Other Functions and Expressions....284
ABS Function....285
Example of ABS....285
ACOS Function....286
Examples of ACOS....286
ADD_MONTHS Function....287
Examples of ADD_MONTHS....287
ASCII Function....288
Considerations for ASCII....288
Example of ASCII....288
ASIN Function....289
Examples of ASIN....289
ATAN Function....290
Examples of ATAN....290
ATAN2 Function....291
Example of ATAN2....291
AUTHNAME Function....292
Considerations for AUTHNAME....292
Example of AUTHNAME....292
AVG Function....293
Considerations for AVG....293
Examples of AVG....293
BITAND Function....295
Considerations for BITAND....295
Restrictions for BITAND....295
Examples of BITAND....295
CASE (Conditional) Expression....296
Considerations for CASE....297
Examples of CASE....297
CAST Expression....299
Considerations for CAST....299
Valid Conversions for CAST....299
Examples of CAST....300
CEILING Function....301
Example of CEILING....301
CHAR Function....302
Considerations for CHAR....302
Example of CHAR....302
CHAR_LENGTH Function....303
Considerations for CHAR_LENGTH....303
Examples of CHAR_LENGTH....303
COALESCE Function....304
Example of COALESCE....304
CODE_VALUE Function....305
Example of CODE_VALUE Function....305

CONCAT Function....306
Concatenation Operator (||)....306
Considerations for CONCAT....306
Examples of CONCAT....306
CONVERTTOHEX Function....308
Considerations for CONVERTTOHEX....308
Examples of CONVERTTOHEX....308
CONVERTTIMESTAMP Function....310
Considerations for CONVERTTIMESTAMP....310
Examples of CONVERTTIMESTAMP....310
COS Function....311
Example of COS....311
COSH Function....312
Example of COSH....312
COUNT Function....313
Considerations for COUNT....313
Examples of COUNT....313
CURRENT Function....315
Example of CURRENT....315
CURRENT_DATE Function....316
Examples of CURRENT_DATE....316
CURRENT_TIME Function....317
Example of CURRENT_TIME....317
CURRENT_TIMESTAMP Function....318
Example of CURRENT_TIMESTAMP....318
CURRENT_USER Function....319
Considerations for CURRENT_USER....319
Example of CURRENT_USER....319
DATE_ADD Function....320
Examples of DATE_ADD....320
DATE_SUB Function....321
Examples of DATE_SUB....321
DATEADD Function....322
Examples of DATEADD....322
DATEDIFF Function....323
Examples of DATEDIFF....323
DATEFORMAT Function....324
Considerations for DATEFORMAT....324
Examples of DATEFORMAT....324
DATE_PART Function (of an Interval)....325
Examples of DATE_PART....325
DATE_PART Function (of a Timestamp)....326
Examples of DATE_PART....326
DATE_TRUNC Function....327
Examples of DATE_TRUNC....327
DAY Function....328
Example of DAY....328
DAYNAME Function....329
Considerations for DAYNAME....329
Example of DAYNAME....329
DAYOFMONTH Function....330
Examples of DAYOFMONTH....330
DAYOFWEEK Function....331
Example of DAYOFWEEK....331
DAYOFYEAR Function....332

Example of DAYOFYEAR....332
DECODE Function....333
Considerations for DECODE....333
Examples of DECODE....334
DEGREES Function....336
Examples of DEGREES....336
DIFF1 Function....337
Considerations for DIFF1....337
Examples of DIFF1....337
DIFF2 Function....339
Considerations for DIFF2....339
Examples of DIFF2....339
EXP Function....341
Examples of EXP....341
EXPLAIN Function....342
Considerations for EXPLAIN Function....342
Examples of EXPLAIN Function....344
EXTRACT Function....345
Examples of EXTRACT....345
FLOOR Function....346
Examples of FLOOR....346
HOUR Function....347
Example of HOUR....347
INSERT Function....348
Examples of INSERT....348
ISNULL Function....349
Examples of ISNULL....349
JULIANTIMESTAMP Function....350
Considerations for JULIANTIMESTAMP....350
Examples of JULIANTIMESTAMP....350
LASTNOTNULL Function....351
Example of LASTNOTNULL....351
LCASE Function....352
Example of LCASE....352
LEFT Function....353
Examples of LEFT....353
LOCATE Function....354
Considerations for LOCATE....354
Examples of LOCATE....354
LOG Function....355
Example of LOG....355
LOG10 Function....356
Example of LOG10....356
LOWER Function....357
Considerations for LOWER....357
Example of LOWER....357
LPAD Function....358
Examples of LPAD....358
LTRIM Function....359
Considerations for LTRIM....359
Example of LTRIM....359
MAX/MAXIMUM Function....360
Considerations for MAX/MAXIMUM....360
Example of MAX/MAXIMUM....360
MIN Function....361

Considerations for MIN....361
Example of MIN....361
MINUTE Function....362
Example of MINUTE....362
MOD Function....363
Example of MOD....363
MONTH Function....364
Example of MONTH....364
MONTHNAME Function....365
Considerations for MONTHNAME....365
Example of MONTHNAME....365
MOVINGAVG Function....366
Example of MOVINGAVG....366
MOVINGCOUNT Function....367
Considerations for MOVINGCOUNT....367
Example of MOVINGCOUNT....367
MOVINGMAX Function....368
Example of MOVINGMAX....368
MOVINGMIN Function....369
Example of MOVINGMIN....369
MOVINGSTDDEV Function....370
Example of MOVINGSTDDEV....370
MOVINGSUM Function....372
Example of MOVINGSUM....372
MOVINGVARIANCE Function....373
Example of MOVINGVARIANCE....373
NULLIF Function....375
Example of NULLIF....375
NULLIFZERO Function....376
Examples of NULLIFZERO....376
NVL Function....377
Examples of NVL....377
OCTET_LENGTH Function....378
Considerations for OCTET_LENGTH....378
Example of OCTET_LENGTH....378
OFFSET Function....379
Example of OFFSET....379
PI Function....380
Example of PI....380
POSITION Function....381
Considerations for POSITION....381
Examples of POSITION....381
POWER Function....382
Examples of POWER....382
QUARTER Function....383
Example of QUARTER....383
RADIANS Function....384
Examples of RADIANS....384
RANK/RUNNINGRANK Function....385
Considerations for RANK/RUNNINGRANK....385
Examples of RANK/RUNNINGRANK....385
REPEAT Function....388
Example of REPEAT....388
REPLACE Function....389
Example of REPLACE....389

RIGHT Function....390
Examples of RIGHT....390
ROUND Function....391
Examples of ROUND....391
ROWS SINCE Function....392
Considerations for ROWS SINCE....392
Examples of ROWS SINCE....392
ROWS SINCE CHANGED Function....394
Considerations for ROWS SINCE CHANGED....394
Examples of ROWS SINCE CHANGED....394
RPAD Function....395
Examples of RPAD Function....395
RTRIM Function....396
Considerations for RTRIM....396
Example of RTRIM....396
RUNNINGAVG Function....397
Considerations for RUNNINGAVG....397
Example of RUNNINGAVG....397
RUNNINGCOUNT Function....398
Considerations for RUNNINGCOUNT....398
Example of RUNNINGCOUNT....398
RUNNINGMAX Function....399
Example of RUNNINGMAX....399
RUNNINGMIN Function....400
Example of RUNNINGMIN....400
RUNNINGRANK Function....400
RUNNINGSTDDEV Function....401
Considerations for RUNNINGSTDDEV....401
Examples of RUNNINGSTDDEV....401
RUNNINGSUM Function....402
Example of RUNNINGSUM....402
RUNNINGVARIANCE Function....403
Examples of RUNNINGVARIANCE....403
SECOND Function....404
Example of SECOND....404
SIGN Function....405
Examples of SIGN....405
SIN Function....406
Example of SIN....406
SINH Function....407
Example of SINH....407
SPACE Function....408
Example of SPACE....408
SQRT Function....409
Example of SQRT....409
STDDEV Function....410
Considerations for STDDEV....410
Examples of STDDEV....411
SUBSTRING/SUBSTR Function....412
Alternative Forms....412
Considerations for SUBSTRING/SUBSTR....412
Examples of SUBSTRING/SUBSTR....413
SUM Function....414
Considerations for SUM....414
Example of SUM....414

TAN Function....415
Example of TAN....415
TANH Function....416
Example of TANH....416
THIS Function....417
Considerations for THIS....417
Example of THIS....417
TIMESTAMPADD Function....418
Examples of TIMESTAMPADD....418
TIMESTAMPDIFF Function....419
Examples of TIMESTAMPDIFF....419
TRANSLATE Function....420
TRIM Function....421
Considerations for TRIM....421
Examples of TRIM....421
UCASE Function....422
Considerations for UCASE....422
Examples of UCASE....422
UPPER Function....423
Example of UPPER....423
UPSHIFT Function....424
Examples of UPSHIFT....424
USER Function....425
Considerations for USER....425
Examples of USER....425
VARIANCE Function....426
Considerations for VARIANCE....426
Examples of VARIANCE....427
WEEK Function....428
Example of WEEK....428
YEAR Function....429
Example of YEAR....429
ZEROIFNULL Function....430
Example of ZEROIFNULL....430
7 OLAP Functions....431
Considerations for Window Functions....431
ORDER BY Clause Supports Expressions For OLAP Functions....431
Limitations for Window Functions....432
AVG Window Function....433
Examples of AVG Window Function....433
COUNT Window Function....434
Examples of COUNT Window Function....434
DENSE_RANK Window Funtion....435
Examples of DENSE_RANK Window Function....435
MAX Window Function....435
Examples of MAX Window Function....436
MIN Window Function....436
Examples of MIN Window Function....437
RANK Window Function....437
Examples of RANK Window Function....438
ROW_NUMBER Window Function....438
Examples of ROW_NUMBER Window Function....438
STDDEV Window Function....438
Examples of STDDEV....439

SUM Window Function....439
Examples of SUM Window Function....440
VARIANCE Window Function....440
Examples of VARIANCE Window Function....441
8 SQL Runtime Statistics....442
PERTABLE and OPERATOR Statistics....442
Adaptive Statistics Collection....442
Retrieving SQL Runtime Statistics....443
Using the GET STATISTICS Command....443
Displaying SQL Runtime Statistics....447
Examples of Displaying SQL Runtime Statistics....451
Using the Parent Query ID....455
Child Query ID....456
Gathering Statistics About RMS....457
Using the QUERYID_EXTRACT Function....459
Syntax of QUERYID_EXTRACT....459
Examples of QUERYID_EXTRACT....459
Statistics for Each Fragment-Instance of an Active Query....460
Syntax of STATISTICS Table-Valued Function....460
Considerations For Obtaining Statistics For Each Fragment-Instance of an Active Query....460
A Reserved Words....462
Reserved Trafodion SQL Identifiers....462
B Control Query Default (CQD) Attributes....466
HBase Environment CQDs....466
HBASE_INTERFACE....466
Hive Environment CQDs....466
HIVE_MAX_STRING_LENGTH....466
Managing Histograms....466
CACHE_HISTOGRAMS_REFRESH_INTERVAL....466
HIST_NO_STATS_REFRESH_INTERVAL....467
HIST_PREFETCH....467
HIST_ROWCOUNT_REQUIRING_STATS....468
Optimizer....468
JOIN_ORDER_BY_USER....468
MDAM_SCAN_METHOD....469
SUBQUERY_UNNESTING....469
Managing Schemas....470
SCHEMA....470
Transaction Control and Locking....470
BLOCK_TO_PREVENT_HALLOWEEN....470
UPD_ORDERED....471
C Limits....472
Index....473

About This Document

This manual describes reference information about the syntax of SQL statements, functions, and other SQL language elements supported by the Trafodion project's database software.

Trafodion SQL statements and utilities are entered interactively or from script files using a client-based tool, such as the Trafodion Command Interface (TrafCI). To install and configure a client application that enables you to connect to and use a Trafodion database, see the *Trafodion Client Installation Guide*.

NOTE: In this manual, SQL language elements, statements, and clauses within statements are based on the ANSI SQL:1999 standard.

Intended Audience

This manual is intended for database administrators and application programmers who are using SQL to read, update, and create Trafodion SQL tables, which map to HBase tables, and to access native HBase and Hive tables.

You should be familiar with structured query language (SQL) and with the American National Standard Database Language SQL:1999.

New and Changed Information in This Edition

This edition includes updates for these new features:

New Feature	Location in the Manual
On Line Analytical Process (OLAP) window functions	• "OLAP Functions" (page 431)
Ability to cancel DDL, update statistics, and additional child query operations in addition to DML statements	• "CONTROL QUERY CANCEL Statement" (page 47)
Authorization required to run the CONTROL QUERY CANCEL Statement	• "CONTROL QUERY CANCEL Statement" (page 47)
Ability to grant privileges on behalf of a role using the GRANTED BY clause.	• "GRANT COMPONENT PRIVILEGE Statement" (page 114) • "GRANT Statement" (page 111)
Authorization required for all SHOWDDL commands	• "SHOWDDL Statement" (page 160) • "SHOWDDL SCHEMA Statement" (page 163)
Ability to display the DDL syntax of a library object using the SHOWDDL LIBRARY command	• "SHOWDDL Statement" (page 160)
Listing of HBase objects using the GET HBASE OBJECTS command through an SQL interface	• "GET HBASE OBJECTS Statement" (page 107)

Document Organization

Chapter or Appendix	Description
Chapter 1: "Introduction" (page 21)	Introduces Trafodion SQL and covers topics such as data consistency, transaction management, and ANSI compliance.
Chapter 2: "SQL Statements" (page 30)	Describes the SQL statements supported by Trafodion SQL.
Chapter 3: "SQL Utilities" (page 176)	Describes the SQL utilities supported by Trafodion SQL.

Chapter or Appendix	Description
Chapter 4: "SQL Language Elements" (page 192)	Describes parts of the language, such as database objects, data types, expressions, identifiers, literals, and predicates, which occur within the syntax of Trafodion SQL statements.
Chapter 5: "SQL Clauses" (page 256)	Describes clauses used by Trafodion SQL statements.
Chapter 6: "SQL Functions and Expressions" (page 278)	Describes specific functions and expressions that you can use in Trafodion SQL statements.
Chapter 8: "SQL Runtime Statistics" (page 442)	Describes how to gather statistics for active queries or for the Runtime Management System (RMS) and describes the RMS counters that are returned.
Chapter 7: "OLAP Functions" (page 431)	Describes specific on line analytical processing functions.
Appendix A: "Reserved Words" (page 462)	Lists the words that are reserved in Trafodion SQL.
Appendix B: "Control Query Default (CQD) Attributes" (page 466)	Describes the Control Query Default (CQD) attributes that are supported in a Trafodion SQL environment.
Appendix C: "Limits" (page 472)	Describes limits in Trafodion SQL.

Notation Conventions

General Syntax Notation

This list summarizes the notation conventions for syntax presentation in this manual.

UPPERCASE LETTERS

Uppercase letters indicate keywords and reserved words. Type these items exactly as shown. Items not enclosed in brackets are required. For example:

SELECT

Italic Letters

Italic letters, regardless of font, indicate variable items that you supply. Items not enclosed in brackets are required. For example:

file-name

Computer Type

Computer type letters within text indicate case-sensitive keywords and reserved words. Type these items exactly as shown. Items not enclosed in brackets are required. For example:

```
myfile.sh
```

Bold Text

Bold text in an example indicates user input typed at the terminal. For example:

```
ENTER RUN CODE

?123
CODE RECEIVED:      123.00
```

The user must press the Return key after typing the input.

[ ] Brackets

Brackets enclose optional syntax items. For example:

```
DATETIME [start-field TO] end-field
```

A group of items enclosed in brackets is a list from which you can choose one item or none. The items in the list can be arranged either vertically, with aligned brackets on each side of

the list, or horizontally, enclosed in a pair of brackets and separated by vertical lines. For example:

```
DROP VIEW view [RESTRICT]
                    [CASCADE]

DROP VIEW view [ RESTRICT | CASCADE ]
```

{ } Braces

Braces enclose required syntax items. For example:

```
FROM { grantee[, grantee]...}
```

A group of items enclosed in braces is a list from which you are required to choose one item. The items in the list can be arranged either vertically, with aligned braces on each side of the list, or horizontally, enclosed in a pair of braces and separated by vertical lines. For example:

```
INTERVAL { start-field TO end-field }
         { single-field }

INTERVAL { start-field TO end-field  | single-field }
```

| Vertical Line

A vertical line separates alternatives in a horizontal list that is enclosed in brackets or braces. For example:

```
{expression | NULL}
```

… Ellipsis

An ellipsis immediately following a pair of brackets or braces indicates that you can repeat the enclosed sequence of syntax items any number of times. For example:

```
ATTRIBUTE[S] attribute [, attribute]...

{, sql-expression}...
```

An ellipsis immediately following a single syntax item indicates that you can repeat that syntax item any number of times. For example:

```
expression-n…
```

Punctuation

Parentheses, commas, semicolons, and other symbols not previously described must be typed as shown. For example:

```
DAY (datetime-expression)

@script-file
```

Quotation marks around a symbol such as a bracket or brace indicate that the symbol is a required character that you must type as shown. For example:

```
"[" ANY N "]" | "[" FIRST N "]"
```

According to the previous syntax, you must include square brackets around ANY and FIRST clauses (for example, `[ANY 10]` or `[FIRST 5]`). Do not include the quotation marks.

Item Spacing

Spaces shown between items are required unless one of the items is a punctuation symbol such as a parenthesis or a comma. For example:

```
DAY (datetime-expression)

DAY(datetime-expression)
```

If no space exists between two items, spaces are not permitted. In this example, no spaces are permitted between the period and any other items:

```
myfile.sh
```

Line Spacing

If the syntax of a command is too long to fit on a single line, each continuation line is indented three spaces and is separated from the preceding line by a blank line. This spacing distinguishes items in a continuation line from items in a vertical list of selections. For example:

```
match-value [NOT] LIKE pattern

   [ESCAPE esc-char-expression]
```

Publishing History

Part Number	Product Version	Publication Date
T775-110-001	Trafodion Release 1.1.0	April 2015
T775-100-001	Trafodion Release 1.0.0	January 2015
T775-090-001	Trafodion Release 0.9.0 Beta	October 2014
T775-080-003	Trafodion Release 0.8.0 Beta This edition of the manual includes updates to address Launchpad bug 1354228. See the "CREATE TABLE Statement" (page 69).	August 2014
T775-080-002	Trafodion Release 0.8.0 Beta This edition of the manual includes updates to address Launchpad bug 1352479. See the "SELECT Statement" (page 138).	August 2014
T775-080-001	Trafodion Release 0.8.0 Beta	June 2014

We Encourage Your Comments

The Trafodion community encourages your comments concerning this document. We are committed to providing documentation that meets your needs. Send any errors found, suggestions for improvement, or compliments to:

trafodion-documentation@lists.launchpad.net

Include the document title, part number, and any comment, error found, or suggestion for improvement you have concerning this document.

1 Introduction

The Trafodion SQL database software allows you to use SQL statements, which comply closely to ANSI SQL:1999, to access data in Trafodion SQL tables, which map to HBase tables, and to access native HBase tables and Hive tables.

This introduction describes:

- "SQL Language"
- "Using Trafodion SQL to Access HBase Tables"
- "Using Trafodion SQL to Access Hive Tables"
- "Data Consistency and Access Options"
- "Transaction Management"
- "ANSI Compliance and Trafodion SQL Extensions"
- "Trafodion SQL Error Messages"

Other sections of this manual describe the syntax and semantics of individual statements, commands, and language elements.

SQL Language

The SQL language consists of statements and other language elements that you can use to access SQL databases. For descriptions of individual SQL statements, see Chapter 2: "SQL Statements" (page 30).

SQL language elements are part of statements and commands and include data types, expressions, functions, identifiers, literals, and predicates. For more information, see Chapter 4: "SQL Language Elements" (page 192) and Chapter 5: "SQL Clauses" (page 256). For information on specific functions and expressions, see Chapter 6: "SQL Functions and Expressions" (page 278) and Chapter 7: "OLAP Functions" (page 431).

Using Trafodion SQL to Access HBase Tables

You can use Trafodion SQL statements to read, update, and create HBase tables.

- "Initializing the Trafodion Metadata" (page 21)
- "Ways to Access HBase Tables" (page 22)
- "Trafodion SQL Tables Versus Native HBase Tables" (page 23)
- "Supported SQL Statements With HBase Tables" (page 23)

For a list of Control Query Default (CQD) settings for the HBase environment, see "HBase Environment CQDs" (page 466).

Initializing the Trafodion Metadata

Before using SQL statements for the first time to access HBase tables, you will need to initialize the Trafodion metadata. To initialize the Trafodion metadata, run this command:

```
initialize trafodion;
```

Ways to Access HBase Tables

Trafodion SQL supports these ways to access HBase tables:

- "Accessing Trafodion SQL Tables" (page 22)
- "Cell-Per-Row Access to HBase Tables (Technology Preview)" (page 22)
- "Rowwise Access to HBase Tables (Technology Preview)" (page 23)

Accessing Trafodion SQL Tables

A Trafodion SQL table is a relational SQL table generated by a CREATE TABLE statement and mapped to an HBase table. Trafodion SQL tables have regular ANSI names in the catalog TRAFODION. A Trafodion SQL table name can be a fully qualified ANSI name of the form `TRAFODION.schema-name.object-name`.

To access a Trafodion SQL table, specify its ANSI table name in a Trafodion SQL statement, similar to how you would specify an ANSI table name when running SQL statements in a relational database. For example:

```
CREATE TABLE trafodion.sales.odetail
  ( ordernum NUMERIC (6) UNSIGNED NO DEFAULT NOT NULL,
    partnum NUMERIC (4) UNSIGNED NO DEFAULT NOT NULL,
    unit_price NUMERIC (8,2) NO DEFAULT NOT NULL,
    qty_ordered NUMERIC (5) UNSIGNED NO DEFAULT NOT NULL,
    PRIMARY KEY (ordernum, partnum) );

INSERT INTO trafodion.sales.odetail VALUES ( 900000, 7301, 425.00, 100 );

SET SCHEMA trafodion.sales;
SELECT * FROM odetail;
```

For more information about Trafodion SQL tables, see "Trafodion SQL Tables Versus Native HBase Tables" (page 23) and "Tables" (page 254).

Cell-Per-Row Access to HBase Tables (Technology Preview)

NOTE: This is a *Technology Preview (Complete But Not Tested)* feature, meaning that it is functionally complete but has not been tested or debugged. For more information about what *Technology Preview* means, see the Technology Preview Features page on the Trafodion wiki.

To access HBase data using cell-per-row mode, specify the schema `HBASE."_CELL_"` and the full ANSI name of the table as a delimited table name. You can specify the name of any HBase table, regardless of whether it was created through Trafodion SQL. For example:

```
select * from hbase."_CELL_"."TRAFODION.MYSCH.MYTAB";
select * from hbase."_CELL_"."table_created_in_HBase";
```

All tables accessed through this schema have the same column layout:

```
>>invoke hbase."_CELL_"."table_created_in_HBase";
  (
    ROW_ID             VARCHAR(100)   ...
  , COL_FAMILY         VARCHAR(100)   ...
  , COL_NAME           VARCHAR(100)   ...
  , COL_TIMESTAMP      LARGEINT       ...
  , COL_VALUE          VARCHAR(1000)  ...
  )
  PRIMARY KEY (ROW_ID)
>>select * from hbase."_CELL_"."mytab";
```

Rowwise Access to HBase Tables (Technology Preview)

NOTE: This is a *Technology Preview (Complete But Not Tested)* feature, meaning that it is functionally complete but has not been tested or debugged. For more information about what *Technology Preview* means, see the Technology Preview Features page on the Trafodion wiki.

To access HBase data using rowwise mode, specify the schema `HBASE."_ROW_"` and the full ANSI name of the table as a delimited table name. You can specify the name of any HBase table, regardless of whether it was created through Trafodion SQL. For example:

```
select * from hbase."_ROW_"."TRAFODION.MYSCH.MYTAB";
select * from hbase."_ROW_"."table_created_in_HBase";
```

All column values of the row are returned as a single, big varchar:

```
>>invoke hbase."_ROW_"."mytab";
  (
    ROW_ID             VARCHAR(100)    ...
  , COLUMN_DETAILS     VARCHAR(10000)  ...
  )
  PRIMARY KEY (ROW_ID)
>>select * from hbase."_ROW_"."mytab";
```

Trafodion SQL Tables Versus Native HBase Tables

Trafodion SQL tables have many advantages over regular HBase tables:

- They can be made to look like regular, structured SQL tables with fixed columns.
- They support the usual SQL data types supported in relational databases.
- They support compound keys, unlike HBase tables that have a single row key (a string).
- They support indexes.
- They support *salting*, which is a technique of adding a hash value of the row key as a key prefix to avoid hot spots for sequential keys. For the syntax, see the "CREATE TABLE Statement" (page 69).

The problem with Trafodion SQL tables is that they use a fixed format to represent column values, making it harder for native HBase applications to access them. Also, they have a fixed structure, so users lose the flexibility of dynamic columns that comes with HBase.

Supported SQL Statements With HBase Tables

You can use these SQL statements with HBase tables:

- "SELECT Statement" (page 138)
- "INSERT Statement" (page 118)
- "UPDATE Statement" (page 169)
- "DELETE Statement" (page 86)
- "MERGE Statement" (page 123)
- "GET Statement" (page 103)
- "INVOKE Statement" (page 122)
- "ALTER TABLE Statement" (page 36)
- "CREATE INDEX Statement" (page 53)
- "CREATE TABLE Statement" (page 69)
- "CREATE VIEW Statement" (page 81)
- "DROP INDEX Statement" (page 89)

- "DROP TABLE Statement" (page 96)
- "DROP VIEW Statement" (page 97)
- "GRANT Statement" (page 111)
- "REVOKE Statement" (page 130)

Using Trafodion SQL to Access Hive Tables

You can use Trafodion SQL statements to access Hive tables.

- "ANSI Names for Hive Tables" (page 24)
- "Type Mapping From Hive to Trafodion SQL" (page 24)
- "Supported SQL Statements With Hive Tables" (page 24)

For a list of Control Query Default (CQD) settings for the Hive environment, see "Hive Environment CQDs" (page 466).

ANSI Names for Hive Tables

Hive tables appear in the Trafodion Hive ANSI namespace in a special catalog and schema named HIVE.HIVE.

To select from a Hive table named T, specify an implicit or explicit name, such as HIVE.HIVE.T, in a Trafodion SQL statement. This example should work if a Hive table named T has already been defined:

```
set schema hive.hive;
cqd hive_max_string_length '20'; -- creates a more readable display
select * from t; -- implicit table name
set schema trafodion.seabase;
select * from hive.hive.t; -- explicit table name
```

Type Mapping From Hive to Trafodion SQL

Trafodion performs the following data-type mappings:

Hive Type	Trafodion SQL Type
tinyint	smallint
smallint	smallint
int	int
bigint	largeint
string	varchar(*n* bytes) character set utf8[1]
float	real
double	float(54)
timestamp	timestamp(6)[2]

[1] The value *n* is determined by CQD HIVE_MAX_STRING_LENGTH. See "Hive Environment CQDs" (page 466).

[2] Hive supports timestamps with nanosecond resolution (precision of 9). Trafodion SQL supports only microsecond resolution (precision 6).

Supported SQL Statements With Hive Tables

You can use these SQL statements with Hive tables:

- "SELECT Statement" (page 138)
- "LOAD Statement" (page 177)

- GET TABLES (See the "GET Statement" (page 103).)
- "INVOKE Statement" (page 122)

Data Consistency and Access Options

Access options for DML statements affect the consistency of the data that your query accesses.

For any DML statement, you specify access options by using the `FOR option ACCESS` clause and, for a SELECT statement, by using this same clause, you can also specify access options for individual tables and views referenced in the FROM clause.

The possible settings for *option* in a DML statement are:

"READ COMMITTED"	Specifies that the data accessed by the DML statement must be from committed rows.

The SQL default access option for DML statements is READ COMMITTED.

For related information about transactions, see "Transaction Isolation Levels" (page 26).

READ COMMITTED

This option allows you to access only committed data.

The implementation requires that a lock can be acquired on the data requested by the DML statement—but does not actually lock the data, thereby reducing lock request conflicts. If a lock cannot be granted (implying that the row contains uncommitted data), the DML statement request waits until the lock in place is released.

READ COMMITTED provides the next higher level of data consistency (compared to READ UNCOMMITTED). A statement executing with this access option does not allow dirty reads, but both nonrepeatable reads and phantoms are possible.

READ COMMITTED provides sufficient consistency for any process that does not require a repeatable read capability.

READ COMMITTED is the default isolation level.

Transaction Management

A transaction (a set of database changes that must be completed as a group) is the basic recoverable unit in case of a failure or transaction interruption. Transactions are controlled through client tools that interact with the database using ODBC or JDBC. The typical order of events is:

1. Transaction is started.
2. Database changes are made.
3. Transaction is committed.

If, however, the changes cannot be made or if you do not want to complete the transaction, you can abort the transaction so that the database is rolled back to its original state.

This subsection discusses these considerations for transaction management:

- "User-Defined and System-Defined Transactions" (page 26)
- "Rules for DML Statements" (page 26)
- "Effect of AUTOCOMMIT Option" (page 26)
- "Concurrency" (page 26)
- "Transaction Isolation Levels" (page 26)

User-Defined and System-Defined Transactions

User-Defined Transactions

Transactions you define are called *user-defined transactions*. To be sure that a sequence of statements executes successfully or not at all, you can define one transaction consisting of these statements by using the BEGIN WORK statement and COMMIT WORK statement. You can abort a transaction by using the ROLLBACK WORK statement. If AUTOCOMMIT is on, you do not have to end the transaction explicitly as Trafodion SQL will end the transaction automatically. Sometimes an error occurs that requires the user-defined transaction to be aborted. Trafodion SQL will automatically abort the transaction and return an error indicating that the transaction was rolled back.

System-Defined Transactions

In some cases, Trafodion SQL defines transactions for you. These transactions are called *system-defined transactions*. Most DML statements initiate transactions implicitly at the start of execution. See "Implicit Transactions" (page 157). However, even if a transaction is initiated implicitly, you must end a transaction explicitly with the COMMIT WORK statement or the ROLLBACK WORK statement. If AUTOCOMMIT is on, you do not need to end a transaction explicitly.

Rules for DML Statements

If deadlock occurs, the DML statement times out and receives an error.

Effect of AUTOCOMMIT Option

AUTOCOMMIT is an option that can be set in a SET TRANSACTION statement. It specifies whether Trafodion SQL will commit automatically, or roll back if an error occurs, at the end of statement execution. This option applies to any statement for which the system initiates a transaction. See "SET TRANSACTION Statement" (page 157).

If this option is set to ON, Trafodion SQL automatically commits any changes, or rolls back any changes, made to the database at the end of statement execution.

Concurrency

Concurrency is defined by two or more processes accessing the same data at the same time. The degree of concurrency available—whether a process that requests access to data that is already being accessed is given access or placed in a wait queue—depends on the purpose of the access mode (read or update) and the isolation level. Currently, the only isolation level is READ COMMITTED.

Trafodion SQL provides concurrent database access for most operations and controls database access through concurrency control and the mechanism for opening and closing tables. For DML operations, the access option affects the degree of concurrency. See "Data Consistency and Access Options" (page 25).

Transaction Isolation Levels

A transaction has an isolation level that is "READ COMMITTED".

READ COMMITTED

This option, which is ANSI compliant, allows your transaction to access only committed data.

No row locks are acquired when READ COMMITTED is the specified isolation level.

READ COMMITTED provides the next level of data consistency. A transaction executing with this isolation level does not allow dirty reads, but both nonrepeatable reads and phantoms are possible.

READ COMMITTED provides sufficient consistency for any transaction that does not require a repeatable-read capability.

The default isolation level is READ COMMITTED.

ANSI Compliance and Trafodion SQL Extensions

Trafodion SQL complies most closely with Core SQL 99. Trafodion SQL also includes some features from SQL 99 and part of the SQL 2003 standard, and special Trafodion SQL extensions to the SQL language.

Statements and SQL elements in this manual are ANSI compliant unless specified as Trafodion SQL extensions.

ANSI-Compliant Statements

These statements are ANSI compliant, but some might contain Trafodion SQL extensions:

- ALTER TABLE statement
- CALL statement
- COMMIT WORK statement
- CREATE FUNCTION statement
- CREATE PROCEDURE statement
- CREATE ROLE statement
- CREATE SCHEMA statement
- CREATE TABLE statement
- CREATE VIEW statement
- DELETE statement
- DROP FUNCTION statement
- DROP PROCEDURE statement
- DROP ROLE statement
- DROP SCHEMA statement
- DROP TABLE statement
- DROP VIEW statement
- EXECUTE statement
- GRANT statement
- GRANT ROLE statement
- INSERT statement
- MERGE statement
- PREPARE statement
- REVOKE statement
- REVOKE ROLE statement
- ROLLBACK WORK statement
- SELECT statement
- SET SCHEMA statement
- SET TRANSACTION statement
- TABLE statement
- UPDATE statement
- VALUES statement

Statements That Are Trafodion SQL Extensions

These statements are Trafodion SQL extensions to the ANSI standard.

- ALTER LIBRARY statement
- ALTER USER statement
- BEGIN WORK statement
- CONTROL QUERY CANCEL statement
- CONTROL QUERY DEFAULT statement
- CREATE INDEX statement
- CREATE LIBRARY statement
- DROP INDEX statement
- DROP LIBRARY statement
- EXPLAIN statement
- GET statement
- GET HBASE OBJECTS statement
- GET VERSION OF METADATA statement
- GET VERSION OF SOFTWARE statement
- GRANT COMPONENT PRIVILEGE statement
- INVOKE statement
- LOAD statement
- REGISTER USER statement
- REVOKE COMPONENT PRIVILEGE statement
- SHOWCONTROL statement
- SHOWDDL statement
- SHOWDDL SCHEMA statement
- SHOWSTATS statement
- UNLOAD statement
- UNREGISTER USER statement
- UPDATE STATISTICS statement
- UPSERT statement

ANSI-Compliant Functions

These functions are ANSI compliant, but some might contain Trafodion SQL extensions:

- AVG function
- CASE expression
- CAST expression
- CHAR_LENGTH
- COALESCE
- COUNT Function
- CURRENT

- CURRENT_DATE
- CURRENT_TIME
- CURRENT_TIMESTAMP
- CURRENT_USER
- EXTRACT
- LOWER
- MAX
- MIN
- NULLIF
- OCTET_LENGTH
- POSITION
- SESSION_USER
- SUBSTRING
- SUM
- TRIM
- UPPER

All other functions are Trafodion SQL extensions.

Trafodion SQL Error Messages

Trafodion SQL reports error messages and exception conditions. When an error condition occurs, Trafodion SQL returns a message number and a brief description of the condition. For example, Trafodion SQL might display this error message:

```
*** ERROR[1000]  A syntax error occurred.
```

The message number is the SQLCODE value (without the sign). In this example, the SQLCODE value is 1000.

2 SQL Statements

This section describes the syntax and semantics of Trafodion SQL statements.

Trafodion SQL statements are entered interactively or from script files using a client-based tool, such as the Trafodion Command Interface (TrafCI). To install and configure a client application that enables you to connect to and use a Trafodion database, see the *Trafodion Client Installation Guide*.

Categories

The statements are categorized according to their functionality:

- "Data Definition Language (DDL) Statements"
- "Data Manipulation Language (DML) Statements"
- "Transaction Control Statements"
- "Data Control and Security Statements"
- "Stored Procedure and User-Defined Function Statements"
- "Prepared Statements"
- "Control Statements"
- "Object Naming Statements"
- "SHOW, GET, and EXPLAIN Statements"

Data Definition Language (DDL) Statements

Use these DDL statements to create, drop, or alter the definition of a Trafodion SQL schema or object.

NOTE: DDL statements are not currently supported in transactions. That means that you cannot run DDL statements inside a user-defined transaction (BEGIN WORK...COMMIT WORK) or when AUTOCOMMIT is OFF. To run these statements, AUTOCOMMIT must be turned ON (the default) for the session.

"ALTER LIBRARY Statement" (page 34)	Updates the physical filename for a library object in a Trafodion database.
"ALTER TABLE Statement" (page 36)	Changes a table.
"ALTER USER Statement" (page 41)	Changes attributes for a user.
"CREATE FUNCTION Statement" (page 50)	Registers a user-defined function (UDF) written in C as a function within a Trafodion database.
"CREATE INDEX Statement" (page 53)	Creates an index on a table.
"CREATE LIBRARY Statement" (page 56)	Registers a library object in a Trafodion database.
"CREATE PROCEDURE Statement" (page 58)	Registers a Java method as a stored procedure in Java (SPJ) within a Trafodion database.
"CREATE ROLE Statement" (page 66)	Creates a role.
"CREATE SCHEMA Statement" (page 67)	Creates a schema in the database.
"CREATE TABLE Statement" (page 69)	Creates a table.
"CREATE VIEW Statement" (page 81)	Creates a view.
"DROP FUNCTION Statement" (page 88)	Removes a user-defined function (UDF) from the Trafodion database.
"DROP INDEX Statement" (page 89)	Drops an index.

"DROP LIBRARY Statement" (page 90)	Removes a library object from the Trafodion database and also removes the library file referenced by the library object.
"DROP PROCEDURE Statement" (page 92)	Removes a stored procedure in Java (SPJ) from the Trafodion database.
"DROP ROLE Statement" (page 93)	Drops a role.
"DROP SCHEMA Statement" (page 95)	Drops a schema from the database.
"DROP TABLE Statement" (page 96)	Drops a table.
"DROP VIEW Statement" (page 97)	Drops a view.
"REGISTER USER Statement" (page 128)	Registers a user in the SQL database, associating the user's login name with a database username.
"UNREGISTER USER Statement" (page 168)	Removes a database username from the SQL database.

Data Manipulation Language (DML) Statements

Use these DML statements to delete, insert, select, or update rows in one or more tables:

"DELETE Statement" (page 86)	Deletes rows from a table or view.
"INSERT Statement" (page 118)	Inserts data into tables and views.
"MERGE Statement" (page 123)	Either performs an upsert operation (that is, updates a table if the row exists or inserts into a table if the row does not exist) or updates (merges) matching rows from one table to another.
"SELECT Statement" (page 138)	Retrieves data from tables and views.
"TABLE Statement" (page 167)	Equivalent to the query specification `SELECT * FROM table`
"UPDATE Statement" (page 169)	Updates values in columns of a table or view.
"UPSERT Statement" (page 173)	Updates a table if the row exists or inserts into a table if the row does not exist.
"VALUES Statement" (page 175)	Displays the results of the evaluation of the expressions and the results of row subqueries within the row value constructors.

Transaction Control Statements

Use these statements to specify user-defined transactions and to set attributes for the next transaction:

"BEGIN WORK Statement" (page 42)	Starts a transaction.
"COMMIT WORK Statement" (page 46)	Commits changes made during a transaction and ends the transaction.
"ROLLBACK WORK Statement" (page 137)	Undoes changes made during a transaction and ends the transaction.
"SET TRANSACTION Statement" (page 157)	Sets attributes for the next SQL transaction — whether to automatically commit database changes.

Data Control and Security Statements

Use these statements to register users, create roles, and grant and revoke privileges:

"ALTER USER Statement" (page 41)	Changes attributes associated with a user who is registered in the database.
"CREATE ROLE Statement" (page 66)	Creates an SQL role.
"DROP ROLE Statement" (page 93)	Deletes an SQL role.

"GRANT Statement" (page 111)	Grants access privileges on an SQL object to specified users or roles.
"GRANT COMPONENT PRIVILEGE Statement" (page 114)	Grants one or more component privileges to a user or role.
"GRANT ROLE Statement" (page 117)	Grants one or more roles to a user.
"REGISTER USER Statement" (page 128)	Registers a user in the SQL database, associating the user's login name with a database username.
"REVOKE Statement" (page 130)	Revokes access privileges on an SQL object from specified users or roles.
"REVOKE COMPONENT PRIVILEGE Statement" (page 133)	Removes one or more component privileges from a user or role.
"REVOKE ROLE Statement" (page 135)	Removes one or more roles from a user.
"UNREGISTER USER Statement" (page 168)	Removes a database username from the SQL database.

Stored Procedure and User-Defined Function Statements

Use these statements to create and execute stored procedures in Java (SPJs) or create user-defined functions (UDFs) and to modify authorization to access libraries or to execute SPJs or UDFs:

"ALTER LIBRARY Statement" (page 34)	Updates the physical filename for a library object in a Trafodion database.
"CALL Statement" (page 43)	Initiates the execution of a stored procedure in Java (SPJ) in a Trafodion database.
"CREATE FUNCTION Statement" (page 50)	Registers a user-defined function (UDF) written in C as a function within a Trafodion database.
"CREATE LIBRARY Statement" (page 56)	Registers a library object in a Trafodion database.
"CREATE PROCEDURE Statement" (page 58)	Registers a Java method as a stored procedure in Java (SPJ) within a Trafodion database.
"DROP FUNCTION Statement" (page 88)	Removes a user-defined function (UDF) from the Trafodion database.
"DROP LIBRARY Statement" (page 90)	Removes a library object from the Trafodion database and also removes the library file referenced by the library object.
"DROP PROCEDURE Statement" (page 92)	Removes a stored procedure in Java (SPJ) from the Trafodion database.
"GRANT Statement" (page 111)	Grants privileges for accessing a library object or executing an SPJ or UDF to specified users.
"REVOKE Statement" (page 130)	Revokes privileges for accessing a library object or executing an SPJ or UDF from specified users.

Prepared Statements

Use these statements to prepare and execute an SQL statement:

"EXECUTE Statement" (page 98)	Executes an SQL statement previously compiled by a PREPARE statement.
"PREPARE Statement" (page 126)	Compiles an SQL statement for later use with the EXECUTE statement in the same session.

Control Statements

Use these statements to control the execution, default options, plans, and performance of DML statements:

"CONTROL QUERY CANCEL Statement" (page 47)	Cancels an executing query that you identify with a query ID.
"CONTROL QUERY DEFAULT Statement" (page 49)	Changes a default attribute to influence a query plan.

Object Naming Statements

Use this statements to specify default ANSI names for the schema:

"SET SCHEMA Statement" (page 156)	Sets the default ANSI schema for unqualified object names for the current session.

SHOW, GET, and EXPLAIN Statements

Use these statements to display information about database objects or query execution plans:

"EXPLAIN Statement" (page 101)	Displays information contained in the query execution plan.
"GET Statement" (page 103)	Displays the names of database objects, components, component privileges, roles, or users that exist in the Trafodion instance.
"GET HBASE OBJECTS Statement" (page 107)	Displays a list of HBase objects through an SQL interface
"GET VERSION OF METADATA Statement" (page 109)	Displays the version of the metadata in the Trafodion instance and indicates if the metadata is current.
"GET VERSION OF SOFTWARE Statement" (page 110)	Displays the version of the Trafodion software that is installed on the system and indicates if it is current.
"INVOKE Statement" (page 122)	Generates a record description that corresponds to a row in the specified table or view.
"SHOWCONTROL Statement" (page 159)	Displays the CONTROL QUERY DEFAULT attributes in effect.
"SHOWDDL Statement" (page 160)	Describes the DDL syntax used to create an object as it exists in the metadata, or it returns a description of a user, role, or component in the form of a GRANT statement.
"SHOWDDL SCHEMA Statement" (page 163)	Displays the DDL syntax used to create a schema as it exists in the metadata and shows the authorization ID that owns the schema.
"SHOWSTATS Statement" (page 164)	Displays the histogram statistics for one or more groups of columns within a table. These statistics are used to devise optimized access plans.

ALTER LIBRARY Statement

- "Syntax Description of ALTER LIBRARY"
- "Considerations for ALTER LIBRARY"
- "Examples of ALTER LIBRARY"

The ALTER LIBRARY statement updates the physical filename for a library object in a Trafodion database. A library object can be an SPJ's JAR file or a UDF's library file.

ALTER LIBRARY is a Trafodion SQL extension.

NOTE: DDL statements are not currently supported in transactions. That means that you cannot run this statement inside a user-defined transaction (BEGIN WORK...COMMIT WORK) or when AUTOCOMMIT is OFF. To run this statement, AUTOCOMMIT must be turned ON (the default) for the session.

```
ALTER LIBRARY [[catalog-name.]schema-name.]library-name
   FILE library-filename
   [HOST NAME host-name]
   [LOCAL FILE host-filename]
```

Syntax Description of ALTER LIBRARY

[[*catalog-name*.]*schema-name*.]*library-name*

specifies the ANSI logical name of the library object, where each part of the name is a valid SQL identifier with a maximum of 128 characters. Specify the name of a library object that has already been registered in the schema. If you do not fully qualify the library name, Trafodion SQL qualifies it according to the schema of the current session. For more information, see "Identifiers" (page 221) and "Database Object Names" (page 198).

FILE *library-filename*

specifies the full path of the redeployed library file, which either an SPJ's JAR file or a UDF's library file.

HOST NAME *host-name*

specifies the name of the client host machine where the deployed file resides.

LOCAL FILE *host-filename*

specifies the path on the client host machine where the deployed file is stored.

Considerations for ALTER LIBRARY

- HOST NAME and LOCAL FILE are position dependent.

Required Privileges

To issue an ALTER LIBRARY statement, one of the following must be true:

- You are DB__ROOT.
- You are the owner of the library.
- You have the ALTER or ALTER_LIBRARY component privilege for the SQL_OPERATIONS component.

Examples of ALTER LIBRARY

- This ALTER LIBRARY statement updates the JAR file (SPJs) for a library named SALESLIB in the SALES schema:

```
ALTER LIBRARY sales.saleslib FILE Sales2.jar;
```

- This ALTER LIBRARY statement updates the library file (UDFs) for a library named MYUDFS in the default schema:

```
ALTER LIBRARY myudfs FILE $TMUDFLIB;
```

ALTER TABLE Statement

- "Syntax Description of ALTER TABLE"
- "Considerations for ALTER TABLE"
- "Example of ALTER TABLE"

The ALTER TABLE statement changes a Trafodion SQL table. See "Tables" (page 254).

NOTE: DDL statements are not currently supported in transactions. That means that you cannot run this statement inside a user-defined transaction (BEGIN WORK...COMMIT WORK) or when AUTOCOMMIT is OFF. To run this statement, AUTOCOMMIT must be turned ON (the default) for the session.

```
ALTER TABLE name alter-action

alter-action is:
     ADD [COLUMN] column-definition
   | ADD IF NOT EXISTS column-definition
   | ADD [CONSTRAINT constraint-name] table-constraint
   | DROP CONSTRAINT constraint-name [RESTRICT]
   | RENAME TO new-name [CASCADE]
   | DROP COLUMN [IF EXISTS] column-name

column-definition is:
   column-name data-type
   ([DEFAULT default]
      [[CONSTRAINT constraint-name] column-constraint]...)

data-type is:
     CHAR[ACTER] [(length)[CHARACTERS]]
          [CHARACTER SET char-set-name]
          [UPSHIFT] [[NOT] CASESPECIFIC]
   | CHAR[ACTER] VARYING (length)
          [CHARACTER SET char-set-name]
          [UPSHIFT] [[NOT] CASESPECIFIC]
   | VARCHAR (length) [CHARACTER SET char-set-name]
          [UPSHIFT] [[NOT] CASESPECIFIC]
   | NUMERIC [(precision [,scale])] [SIGNED|UNSIGNED]
   | NCHAR [(length) [CHARACTER SET char-set-name]
          [UPSHIFT] [[NOT] CASESPECIFIC]
   | NCHAR VARYING(length) [CHARACTER SET char-set-name]
          [UPSHIFT] [[NOT] CASESPECIFIC]
   | SMALLINT [SIGNED|UNSIGNED]
   | INT[EGER] [SIGNED|UNSIGNED]
   | LARGEINT
   | DEC[IMAL] [(precision [,scale])] [SIGNED|UNSIGNED]
   | FLOAT [(precision)]
   | REAL
   | DOUBLE PRECISION
   | DATE
   | TIME [(time-precision)]
   | TIMESTAMP [(timestamp-precision)]
   | INTERVAL { start-field TO end-field | single-field }

default is:
     literal
   | NULL
   | CURRENT_DATE
   | CURRENT_TIME
   | CURRENT_TIMESTAMP }
```

```
column-constraint is:
   NOT NULL
   | UNIQUE
   | CHECK (condition)
   | REFERENCES ref-spec

table-constraint is:
   UNIQUE (column-list)
   | CHECK (condition)
   | FOREIGN KEY (column-list) REFERENCES ref-spec

ref-spec is:
   referenced-table [(column-list)]

column-list is:
   column-name[, column-name]...
```

Syntax Description of ALTER TABLE

name

specifies the current name of the object. See "Database Object Names" (page 198).

ADD [COLUMN] *column-definition*

adds a column to *table*.

The clauses for the *column-definition* are:

column-name

specifies the name for the new column in the table. *column-name* is an SQL identifier. *column-name* must be unique among column names in the table. If the column name is a Trafodion SQL reserved word, you must delimit it by enclosing it in double quotes. For example: `"sql".myview`. See "Identifiers" (page 221).

data-type

specifies the data type of the values that can be stored in *column-name*. See "Data Types" (page 199). If a default is not specified, NULL is used.

DEFAULT *default*

specifies a default value for the column or specifies that the column does not have a default value. You can declare the default value explicitly by using the DEFAULT clause, or you can enable null to be used as the default by omitting both the DEFAULT and NOT NULL clauses. If you omit the DEFAULT clause and specify NOT NULL, Trafodion SQL returns an error. For existing rows of the table, the added column takes on its default value.

If you set the default to the datetime value CURRENT_DATE, CURRENT_TIME, or CURRENT_TIMESTAMP, Trafodion SQL uses January 1, 1 A.D. 12:00:00.000000 as the default date and time for the existing rows.

For any row that you add after the column is added, if no value is specified for the column as part of the add row operation, the column receives a default value based on the current timestamp at the time the row is added.

[[CONSTRAINT *constraint-name*] *column-constraint*]

specifies a name for the column or table constraint. *constraint-name* must have the same schema as *table* and must be unique among constraint names in its schema. If you omit the schema portions of the name you specify in *constraint-name*, Trafodion SQL expands the constraint name by using the schema for *table*. See "Database Object Names" (page 198).

If you do not specify a constraint name, Trafodion SQL constructs an SQL identifier as the name for the constraint in the schema for *table*. The identifier consists of the fully qualified table name concatenated with a system-generated unique identifier. For example, a constraint on table A.B.C might be assigned a name such as A.B.C_123..._01... .

column-constraint options:

NOT NULL

is a column constraint that specifies that the column cannot contain nulls. If you omit NOT NULL, nulls are allowed in the column. If you specify both NOT NULL and NO DEFAULT, each row inserted in the table must include a value for the column. See "Null" (page 231).

UNIQUE

is a column constraint that specifies that the column cannot contain more than one occurrence of the same value. If you omit UNIQUE, duplicate values are allowed unless the column is part of the PRIMARY KEY. Columns that you define as unique must be specified as NOT NULL.

CHECK (*condition*)

is a constraint that specifies a condition that must be satisfied for each row in the table. See "Search Condition" (page 250). You cannot refer to the CURRENT_DATE, CURRENT_TIME, or CURRENT_TIMESTAMP function in a CHECK constraint, and you cannot use subqueries in a CHECK constraint.

REFERENCES *ref-spec*

specifies a REFERENCES column constraint. The maximum combined length of the columns for a REFERENCES constraint is 2048 bytes.

ref-spec is:

referenced-table [(*column-list*)]

referenced-table is the table referenced by the foreign key in a referential constraint. *referenced-table* cannot be a view. *referenced-table* cannot be the same as *table*. *referenced-table* corresponds to the foreign key in the *table*.

column-list specifies the column or set of columns in the *referenced-table* that corresponds to the foreign key in *table*. The columns in the column list associated with REFERENCES must be in the same order as the columns in the column list associated with FOREIGN KEY. If *column-list* is omitted, the referenced table's PRIMARY KEY columns are the referenced columns.

A table can have an unlimited number of referential constraints, and you can specify the same foreign key in more than one referential constraint, but you must define each referential constraint separately. You cannot create self-referencing foreign key constraints.

ADD [CONSTRAINT *constraint-name*] *table-constraint*

adds a constraint to the table and optionally specifies *constraint-name* as the name for the constraint. The new constraint must be consistent with any data already present in the table.

CONSTRAINT *constraint-name*

specifies a name for the column or table constraint. *constraint-name* must have the same schema as *table* and must be unique among constraint names in its schema. If you omit the schema portions of the name you specify in *constraint-name*, Trafodion SQL expands the constraint name by using the schema for table. See "Database Object Names" (page 198).

If you do not specify a constraint name, Trafodion SQL constructs an SQL identifier as the name for the constraint in the schema for table. The identifier consists of the fully qualified table name concatenated with a system-generated unique identifier. For example, a constraint on table A.B.C might be assigned a name such as A.B.C_123..._01... .

table-constraint options:

UNIQUE (*column-list*)

is a table constraint that specifies that the column or set of columns cannot contain more than one occurrence of the same value or set of values.

column-list cannot include more than one occurrence of the same column. In addition, the set of columns that you specify on a UNIQUE constraint cannot match the set of columns on any other UNIQUE constraint for the table or on the PRIMARY KEY constraint for the table. All columns defined as unique must be specified as NOT NULL.

A UNIQUE constraint is enforced with a unique index. If there is already a unique index on *column-list*, Trafodion SQL uses that index. If a unique index does not exist, the system creates a unique index.

CHECK (*condition*)

is a constraint that specifies a condition that must be satisfied for each row in the table. See "Search Condition" (page 250). You cannot refer to the CURRENT_DATE, CURRENT_TIME, or CURRENT_TIMESTAMP function in a CHECK constraint, and you cannot use subqueries in a CHECK constraint.

FOREIGN KEY (*column-list*) REFERENCES *ref-spec* NOT ENFORCED

is a table constraint that specifies a referential constraint for the table, declaring that a column or set of columns (called a foreign key) in *table* can contain only values that match those in a column or set of columns in the table specified in the REFERENCES clause. However, because NOT ENFORCED is specified, this relationship is not checked.

The two columns or sets of columns must have the same characteristics (data type, length, scale, precision). Without the FOREIGN KEY clause, the foreign key in *table* is the column being defined; with the FOREIGN KEY clause, the foreign key is the column or set of columns specified in the FOREIGN KEY clause. For information about *ref-spec*, see REFERENCES *ref-spec* NOT ENFORCED.

DROP CONSTRAINT *constraint-name* [RESTRICT]

drops a constraint from the table.

If you drop a constraint, Trafodion SQL drops its dependent index if Trafodion SQL originally created the same index. If the constraint uses an existing index, the index is not dropped.

CONSTRAINT *constraint-name*

specifies a name for the column or table constraint. *constraint-name* must have the same schema as *table* and must be unique among constraint names in its schema. If you omit the schema portions of the name you specify in *constraint-name*, Trafodion SQL expands the constraint name by using the schema for table. See "Database Object Names" (page 198).

If you do not specify a constraint name, Trafodion SQL constructs an SQL identifier as the name for the constraint in the schema for table. The identifier consists of the fully qualified table name concatenated with a system-generated unique identifier. For example, a constraint on table A.B.C might be assigned a name such as A.B.C_123..._01... .

RENAME TO *new-name* [CASCADE]

changes the logical name of the object within the same schema.

new-name

specifies the new name of the object after the RENAME TO operation occurs.

CASCADE

specifies that indexes and constraints on the renamed object will be renamed.

ADD IF NOT EXISTS *column-definition*

adds a column to *table* if it does not already exist in the table.

The clauses for the *column-definition* are the same as described in ADD [COLUMN] *column-definition*.

DROP COLUMN [IF EXISTS] *column-name*

drops the specified column from *table*, including the column's data. You cannot drop a primary key column.

Considerations for ALTER TABLE

Effect of Adding a Column on View Definitions

The addition of a column to a table has no effect on existing view definitions. Implicit column references specified by SELECT * in view definitions are replaced by explicit column references when the definition clauses are originally evaluated.

Authorization and Availability Requirements

ALTER TABLE works only on user-created tables.

Required Privileges

To issue an ALTER TABLE statement, one of the following must be true:

- You are DB__ROOT.
- You are the owner of the table.
- You have the ALTER or ALTER_TABLE component privilege for the SQL_OPERATIONS component.

Privileges Needed to Create a Referential Integrity Constraint

To create a referential integrity constraint (that is, a constraint on the table that refers to a column in another table), one of the following must be true:

- You are DB__ROOT.
- You are the owner of the referencing and referenced tables.
- You have these privileges on the referencing and referenced table:
  - For the referencing table, you have the ALTER or ALTER_TABLE component privilege for the SQL_OPERATIONS component.
  - For the referenced table, you have the REFERENCES (or ALL) privilege on the referenced table through your username or through a granted role.

If the constraint refers to the other table in a query expression, you must also have SELECT privileges on the other table.

Example of ALTER TABLE

This example adds a column:

```
ALTER TABLE persnl.project
  ADD COLUMN projlead
    NUMERIC (4) UNSIGNED
```

ALTER USER Statement

- "Syntax Description of ALTER USER"
- "Considerations for ALTER USER"
- "Examples of ALTER USER"

The ALTER USER statement changes attributes associated with a user who is registered in the database.

ALTER USER is a Trafodion SQL extension.

```
ALTER USER database-username alter-action[, alter-action]

alter-action is:
     SET EXTERNAL NAME directory-service-username
   | SET { ONLINE | OFFLINE }
```

Syntax Description of ALTER USER

database-username

is the name of a currently registered database user.

SET EXTERNAL NAME

changes the name that identifies the user in the directory service. This is also the name the user specifies when connecting to the database.

directory-service-username

specifies the new name of the user in the directory service. *directory-service-username* is a regular or delimited case-insensitive identifier. See "Case-Insensitive Delimited Identifiers" (page 221).

SET { ONLINE | OFFLINE }

changes the attribute that controls whether the user is allowed to connect to the database.

ONLINE

specifies that the user is allowed to connect to the database.

OFFLINE

specifies that the user is not allowed to connect to the database.

Considerations for ALTER USER

Only a user with user administrative privileges (that is, a user who has been granted the MANAGE_USERS component privilege) can do the following:

- Set the EXTERNAL NAME for any user
- Set the ONLINE | OFFLINE attribute for any user

Initially, DB__ROOT is the only database user who has been granted the MANAGE_USERS component privilege.

Examples of ALTER USER

- To change a user's external name:

```
ALTER USER ajones SET EXTERNAL NAME "Americas\ArturoJones";
```

- To change a user's attribute to allow the user to connect to the database:

```
ALTER USER ajones SET ONLINE;
```

BEGIN WORK Statement

- "Considerations for BEGIN WORK"
- "Example of BEGIN WORK"

The BEGIN WORK statement enables you to start a transaction explicitly—where the transaction consists of the set of operations defined by the sequence of SQL statements that begins immediately after BEGIN WORK and ends with the next COMMIT or ROLLBACK statement. See "Transaction Management" (page 25). BEGIN WORK will raise an error if a transaction is currently active.

BEGIN WORK is a Trafodion SQL extension.

```
BEGIN WORK
```

Considerations for BEGIN WORK

BEGIN WORK starts a transaction. COMMIT WORK or ROLLBACK WORK ends a transaction.

Example of BEGIN WORK

Group three separate statements—two INSERT statements and an UPDATE statement—that update the database within a single transaction:

```
--- This statement initiates a transaction.
BEGIN WORK;
--- SQL operation complete.

INSERT INTO sales.orders VALUES (125, DATE '2008-03-23',
  DATE '2008-03-30', 75, 7654);
--- 1 row(s) inserted.

INSERT INTO sales.odetail VALUES (125, 4102, 25000, 2);
--- 1 row(s) inserted.

UPDATE invent.partloc SET qty_on_hand = qty_on_hand - 2
  WHERE partnum = 4102 AND loc_code = 'G45';
--- 1 row(s) updated.

--- This statement ends a transaction.
COMMIT WORK;
--- SQL operation complete.
```

CALL Statement

- "Syntax Description of CALL"
- "Considerations for CALL"
- "Examples of CALL"

The CALL statement invokes a stored procedure in Java (SPJ) in a Trafodion SQL database.

```
CALL procedure-ref ([argument-list])

procedure-ref is:
   [[catalog-name.]schema-name.]procedure-name

argument-list is:
   SQL-expression[, SQL-expression]...
```

Syntax Description of CALL

procedure-ref

specifies an ANSI logical name of the form:

```
[[catalog-name.]schema-name.]procedure-name
```

where each part of the name is a valid SQL identifier with a maximum of 128 characters. For more information, see "Identifiers" (page 221) and "Database Object Names" (page 198).

If you do not fully qualify the procedure name, Trafodion SQL qualifies it according to the schema of the current session.

argument-list

accepts arguments for IN, INOUT, or OUT parameters. The arguments consist of SQL expressions, including dynamic parameters, separated by commas:

```
SQL-expression[{, SQL-expression}...]
```

Each expression must evaluate to a value of one of these data types:

- Character value
- Date-time value
- Numeric value

Interval value expressions are disallowed in SPJs. For more information, see "Input Parameter Arguments" (page 44) and "Output Parameter Arguments" (page 44).

Do not specify result sets in the argument list.

Considerations for CALL

Usage Restrictions

You can use a CALL statement as a stand-alone SQL statement in applications or command-line interfaces, such as TrafCI. You cannot use a CALL statement inside a compound statement or with rowsets.

Required Privileges

To issue a CALL statement, one of the following must be true:

- You are DB__ROOT.
- You are the owner of the stored procedure.
- You have the EXECUTE (or ALL) privileges, either directly through your username or through a granted role. For more information, see the "GRANT Statement" (page 111).

When the stored procedure executes, it executes as the Trafodion ID.

Input Parameter Arguments

You pass data to an SPJ by using IN or INOUT parameters. For an IN parameter argument, use one of these SQL expressions:

- Literal
- SQL function (including CASE and CAST expressions)
- Arithmetic or concatenation operation
- Scalar subquery
- Dynamic parameter (for example, ?) in an application
- Named (for example, ?param) or unnamed (for example, ?) parameter in TrafCI

For an INOUT parameter argument, you can use only a dynamic, named, or unnamed parameter. For more information, see "Expressions" (page 211).

Output Parameter Arguments

An SPJ returns values in OUT and INOUT parameters. Output parameter arguments must be dynamic parameters in an application (for example, ?) or named or unnamed parameters in HPDCI (for example, ?param or ?). Each calling application defines the semantics of the OUT and INOUT parameters in its environment.

Data Conversion of Parameter Arguments

Trafodion SQL performs an implicit data conversion when the data type of a parameter argument is compatible with but does not match the formal data type of the stored procedure. For stored procedure input values, the conversion is from the actual argument value to the formal parameter type. For stored procedure output values, the conversion is from the actual output value, which has the data type of the formal parameter, to the declared type of the dynamic parameter.

Null Input and Output

You can pass a null value as input to or output from an SPJ, provided that the corresponding Java data type of the parameter supports nulls. If a null is input or output for a parameter that does not support nulls, Trafodion SQL returns an error.

Transaction Semantics

The CALL statement automatically initiates a transaction if no active transaction exists. However, the failure of a CALL statement does not always automatically abort the transaction.

Examples of CALL

- In TrafCI, execute an SPJ named MONTHLYORDERS, which has one IN parameter represented by a literal and one OUT parameter represented by an unnamed parameter, ?:

```
CALL sales.monthlyorders(3,?);
```

- This CALL statement executes a stored procedure, which accepts one IN parameter (a date literal), returns one OUT parameter (a row from the column, NUM_ORDERS), and returns two result sets:

```
CALL sales.ordersummary('01/01/2001', ?);

NUM_ORDERS
--------------------
                  13

ORDERNUM    NUM_PARTS       AMOUNT           ORDER_DATE  LAST_NAME
----------  --------------  ---------------  ----------  ------------------
```

```
    100210              4         19020.00  2006-04-10  HUGHES
    100250              4         22625.00  2006-01-23  HUGHES
    101220              4         45525.00  2006-07-21  SCHNABL
    200300              3         52000.00  2006-02-06  SCHAEFFER
    200320              4          9195.00  2006-02-17  KARAJAN
    200490              2          1065.00  2006-03-19  WEIGL
      .
      .
      .

--- 13 row(s) selected.

ORDERNUM    PARTNUM   UNIT_PRICE    QTY_ORDERED PARTDESC
----------  --------  ------------  ----------  ------------------

    100210      2001       1100.00           3  GRAPHIC PRINTER,M1
    100210      2403        620.00           6  DAISY PRINTER,T2
    100210       244       3500.00           3  PC GOLD, 30 MB
    100210      5100        150.00          10  MONITOR BW, TYPE 1
    100250      6500         95.00          10  DISK CONTROLLER
    100250      6301        245.00          15  GRAPHIC CARD, HR
      .
      .
      .

--- 70 row(s) selected.

--- SQL operation complete.
```

COMMIT WORK Statement

- "Considerations for COMMIT WORK"
- "Example of COMMIT WORK"

The COMMIT WORK statement commits any changes to objects made during the current transaction and ends the transaction. See "Transaction Management" (page 25).

```
COMMIT [WORK]
```

WORK is an optional keyword that has no effect.

COMMIT WORK issued outside of an active transaction generates error 8605.

Considerations for COMMIT WORK

BEGIN WORK starts a transaction. COMMIT WORK or ROLLBACK WORK ends a transaction.

Example of COMMIT WORK

Suppose that your application adds information to the inventory. You have received 24 terminals from a new supplier and want to add the supplier and update the quantity on hand. The part number for the terminals is 5100, and the supplier is assigned supplier number 17. The cost of each terminal is $800.

The transaction must add the order for terminals to PARTSUPP, add the supplier to the SUPPLIER table, and update QTY_ON_HAND in PARTLOC. After the INSERT and UPDATE statements execute successfully, you commit the transaction, as shown:

```
-- This statement initiates a transaction.
BEGIN WORK;
--- SQL operation complete.

-- This statement inserts a new entry into PARTSUPP.
INSERT INTO invent.partsupp
VALUES (5100, 17, 800.00, 24);
--- 1 row(s) inserted.

-- This statement inserts a new entry into SUPPLIER.
INSERT INTO invent.supplier
VALUES (17, 'Super Peripherals','751 Sanborn Way',
  'Santa Rosa', 'California', '95405');
--- 1 row(s) inserted.

-- This statement updates the quantity in PARTLOC.
UPDATE invent.partloc
SET qty_on_hand = qty_on_hand + 24
WHERE partnum = 5100 AND loc_code = 'G43';
--- 1 row(s) updated.

-- This statement ends a transaction.
COMMIT WORK;
--- SQL operation complete.
```

CONTROL QUERY CANCEL Statement

The CONTROL QUERY CANCEL statement cancels an executing query that you identify with a query ID. You can execute the CONTROL QUERY CANCEL statement in a client-based tool like TrafCI or through any ODBC or JDBC application.

CONTROL QUERY CANCEL is a Trafodion SQL extension.

```
CONTROL QUERY CANCEL QID query-id [COMMENT 'comment-text']
```

Syntax Description of CONTROL QUERY CANCEL

query-id

specifies the query ID of an executing query, which is a unique identifier generated by the SQL compiler.

'*comment-text*'

specifies an optional comment to be displayed in the canceled query's error message.

Considerations for CONTROL QUERY CANCEL

Benefits of CONTROL QUERY CANCEL

For many queries, the CONTROL QUERY CANCEL statement allows the termination of the query without stopping the master executor process (MXOSRVR). This type of cancellation has these benefits over standard ODBC/JDBC cancel methods:

- An ANSI-defined error message is returned to the client session, and SQLSTATE is set to HY008.
- Important cached objects persist after the query is canceled, including the master executor process and its compiler, the compiled statements cached in the master, and the compiler's query cache and its cached metadata and histograms.
- The client does not need to reestablish its connection, and its prepared statements are preserved.
- When clients share connections using a middle-tier application server, the effects of canceling one client's executing query no longer affect other clients sharing the same connection.

Restrictions on CONTROL QUERY CANCEL

Some executing queries may not respond to a CONTROL QUERY CANCEL statement within a 60-second interval. For those queries, Trafodion SQL stops their ESP processes if there are any. If this action allows the query to be canceled, you will see all the benefits listed above.

If the executing query does not terminate within 120 seconds after the CONTROL QUERY CANCEL statement is issued, Trafodion SQL stops the master executor process, terminating the query and generating a lost connection error. In this case, you will not see any of the benefits listed above. Instead, you will lose your connection and will need to reconnect and re-prepare the query. This situation often occurs with the CALL, DDL, and utility statements and rarely with other statements.

The CONTROL QUERY CANCEL statement does not work with these statements:

- Unique queries, which operate on a single row and a single partition
- Queries that are not executing, such as a query that is being compiled
- CONTROL QUERY DEFAULT, BEGIN WORK, COMMIT WORK, ROLLBACK WORK, and EXPLAIN statements
- Statically compiled metadata queries
- Queries executed in anomalous conditions, such as queries without runtime statistics or without a query ID

Required Privileges

To issue a CONTROL QUERY CANCEL statement, one of the following must be true:

- You are DB__ROOT.
- You own (that is, issued) the query.
- You have the QUERY_CANCEL component privilege for the SQL_OPERATIONS component.

Example of CONTROL QUERY CANCEL

This CONTROL QUERY CANCEL statement cancels a specified query and provides a comment concerning the cancel operation:

```
control query cancel qid MXID11000010941212288634364991407000000003806U3333300_156016_S1 comment 'Query is
consuming too many resources.';
```

In a separate session, the client that issued the query will see this error message indicating that the query has been canceled:

```
>>execute s1;

*** ERROR[8007] The operation has been canceled. Query is consuming too many resources.
```

CONTROL QUERY DEFAULT Statement

The CONTROL QUERY DEFAULT statement changes the default settings for the current process. You can execute the CONTROL QUERY DEFAULT statement in a client-based tool like TrafCI or through any ODBC or JDBC application.

CONTROL QUERY DEFAULT is a Trafodion SQL extension.

```
{ CONTROL QUERY DEFAULT | CQD } control-default-option

control-default-option is:

attribute {'attr-value' | RESET}
```

Syntax Description of CONTROL QUERY DEFAULT

attribute

is a character string that represents an attribute name. For descriptions of these attributes, see "Control Query Default (CQD) Attributes" (page 466).

attr-value

is a character string that specifies an attribute value. You must specify *attr-value* as a quoted string—even if the value is a number.

RESET

specifies that the attribute that you set by using a CONTROL QUERY DEFAULT statement in the current session is to be reset to the value or values in effect at the start of the current session.

Considerations for CONTROL QUERY DEFAULT

Scope of CONTROL QUERY DEFAULT

The result of the execution of a CONTROL QUERY DEFAULT statement stays in effect until the current process terminates or until the execution of another statement for the same attribute overrides it. CQDs are applied at compile time, so CQDs do not affect any statements that are already prepared. For example:

```
PREPARE x FROM SELECT * FROM t;
CONTROL QUERY DEFAULT SCHEMA 'myschema';
EXECUTE x;                                          -- uses the default schema SEABASE
SELECT * FROM t2;                                   -- uses MYSCHEMA;
PREPARE y FROM SELECT * FROM t3;
CONTROL QUERY DEFAULT SCHEMA 'seabase';
EXECUTE y;                                          -- uses MYSCHEMA;
```

Examples of CONTROL QUERY DEFAULT

- Change the maximum supported length of the column names to 200 for the current process:

  ```
  CONTROL QUERY DEFAULT HBASE_MAX_COLUMN_NAME_LENGTH '200';
  ```

- Reset the HBASE_MAX_COLUMN_NAME_LENGTH attribute to its initial value in the current process:

  ```
  CONTROL QUERY DEFAULT HBASE_MAX_COLUMN_NAME_LENGTH RESET;
  ```

CREATE FUNCTION Statement

- "Syntax Description of CREATE FUNCTION"
- "Considerations for CREATE FUNCTION"
- "Examples of CREATE FUNCTION"

The CREATE FUNCTION statement registers a user-defined function (UDF) written in C as a function within a Trafodion database. Currently, Trafodion supports the creation of *scalar UDFs*, which return a single value or row when invoked. Scalar UDFs are invoked as SQL expressions in the SELECT list or WHERE clause of a SELECT statement.

NOTE: DDL statements are not currently supported in transactions. That means that you cannot run this statement inside a user-defined transaction (BEGIN WORK...COMMIT WORK) or when AUTOCOMMIT is OFF. To run this statement, AUTOCOMMIT must be turned ON (the default) for the session.

```
CREATE FUNCTION function-ref ([parameter-declaration[, parameter-declaration]...])
   {RETURN | RETURNS} (return-parameter-declaration[, return-parameter-declaration]...)
   EXTERNAL NAME 'character-string-literal'
   LIBRARY [[catalog-name.]schema-name.]library-name
   [LANGUAGE C]
   [PARAMETER STYLE SQL]
   [NO SQL]
   [NOT  DETERMINISTIC | DETERMINISTIC]
   [FINAL CALL | NO FINAL CALL]
   [NO STATE AREA | STATE AREA size]
   [NO PARALLELISM | ALLOW ANY PARALLELISM]

function-ref is:
   [[catalog-name.]schema-name.]function-name

parameter-declaration is:
   [IN] [sql-parameter-name] sql-datatype

return-parameter-declaration is:
   [OUT] [sql-parameter-name] sql-datatype
```

Syntax Description of CREATE FUNCTION

function-ref ([*parameter-declaration*[, *parameter-declaration*]...])

specifies the name of the function and any SQL parameters that correspond to the signature of the external function.

function-ref

specifies an ANSI logical name of the form:

[[*catalog-name*.]*schema-name*.]*function-name*

where each part of the name is a valid SQL identifier with a maximum of 128 characters. For more information, see "Identifiers" (page 221) and "Database Object Names" (page 198).

Specify a name that is unique and does not exist for any procedure or function in the same schema.

If you do not fully qualify the function name, Trafodion SQL qualifies it according to the schema of the current session.

parameter-declaration

specifies an SQL parameter that corresponds to the signature of the external function:

[IN] [*sql-parameter-name*] *sql-datatype*

IN

specifies that the parameter passes data to the function.

sql-parameter-name

specifies an SQL identifier for the parameter. For more information, see "Identifiers" (page 221).

sql-datatype

specifies an SQL data type that corresponds to the data type of the parameter in the signature of the external function. *sql-datatype* is one of the supported SQL data types in Trafodion. See "Data Types" (page 199).

{RETURN | RETURNS} (*return-parameter-declaration*[, *return-parameter-declaration*]...)

specifies the type of output of the function.

return-parameter-declaration

specifies an SQL parameter for an output value:

```
[OUT] [sql-parameter-name] sql-datatype
```

OUT

specifies that the parameter accepts data from the function.

sql-parameter-name

specifies an SQL identifier for the return parameter. For more information, see "Identifiers" (page 221).

sql-datatype

specifies an SQL data type for the return parameter. *sql-datatype* is one of the supported SQL data types in Trafodion. See "Data Types" (page 199).

EXTERNAL NAME '*method-name*'

specifies the case-sensitive name of the external function's method.

LIBRARY [[*catalog-name*.]*schema-name*.]*library-name*

specifies the ANSI logical name of a library containing the external function. If you do not fully qualify the library name, Trafodion SQL qualifies it according to the schema of the current session.

LANGUAGE C

specifies that the external function is written in the C language. This clause is optional.

PARAMETER STYLE SQL

specifies that the run-time conventions for arguments passed to the external function are those of the SQL language. This clause is optional.

NO SQL

specifies that the function does not perform SQL operations. This clause is optional.

DETERMINISTIC | NOT DETERMINISTIC

specifies whether the function always returns the same values for OUT parameters for a given set of argument values (DETERMINISTIC, the default behavior) or does not return the same values (NOT DETERMINISTIC). If the function is deterministic, Trafodion SQL is not required to execute the function each time to produce results; instead, Trafodion SQL caches the results and reuses them during subsequent executions, thus optimizing the execution.

FINAL CALL | NO FINAL CALL

specifies whether or not a final call is made to the function. A final call enables the function to free up system resources. The default is FINAL CALL.

NO STATE AREA | STATE AREA *size*

specifies whether or not a state area is allocated to the function. *size* is an integer denoting memory in bytes. Acceptable values range from 0 to 16000. The default is NO STATE AREA.

NO PARALLELISM | ALLOW ANY PARALLELISM

specifies whether or not parallelism is applied when the function is invoked. The default is ALLOW ANY PARALLELISM.

Considerations for CREATE FUNCTION

Required Privileges

To issue a CREATE FUNCTION statement, one of the following must be true:

- You are DB__ROOT.
- You are creating the function in a shared schema, and you have the USAGE (or ALL) privilege on the library that will be used in the creation of the function. The USAGE privilege provides you with read access to the library's underlying library file.
- You are the private schema owner and have the USAGE (or ALL) privilege on the library that will be used in the creation of the function. The USAGE privilege provides you with read access to the library's underlying library file.
- You have the CREATE or CREATE_ROUTINE component level privilege for the SQL_OPERATIONS component and have the USAGE (or ALL) privilege on the library that will be used in the creation of the function. The USAGE privilege provides you with read access to the library's underlying library file.

NOTE: In this case, if you create a function in a private schema, it will be owned by the schema owner.

Examples of CREATE FUNCTION

- This CREATE FUNCTION statement creates a function that adds two integers:

```
create function add2 (int, int)
       returns (total_value int)
       external name 'add2'
       library myudflib;
```

- This CREATE FUNCTION statement creates a function that returns the minimum, maximum, and average values of five input integers:

```
create function mma5 (int, int, int, int, int)
       returns (min_value int, max_value int, avg_value int)
       external name 'mma5'
       library myudflib;
```

- This CREATE FUNCTION statement creates a function that reverses an input string of at most 32 characters:

```
create function reverse (varchar(32))
       returns (reversed_string varchar(32))
       external name 'reverse'
       library myudflib;
```

CREATE INDEX Statement

- "Syntax Description of CREATE INDEX"
- "Considerations for CREATE INDEX"
- "Examples of CREATE INDEX"

The CREATE INDEX statement creates an SQL index based on one or more columns of a table or table-like object. The CREATE VOLATILE INDEX statement creates an SQL index with a lifespan that is limited to the SQL session that the index is created. Volatile indexes are dropped automatically when the session ends. See "Indexes" (page 222).

CREATE INDEX is a Trafodion SQL extension.

NOTE: DDL statements are not currently supported in transactions. That means that you cannot run this statement inside a user-defined transaction (BEGIN WORK...COMMIT WORK) or when AUTOCOMMIT is OFF. To run this statement, AUTOCOMMIT must be turned ON (the default) for the session.

```
CREATE [VOLATILE] INDEX index ON table
   (column-name [ASC[ENDING] | DESC[ENDING]]
     [,column-name [ASC[ENDING] | DESC[ENDING]]]...)
   [HBASE_OPTIONS (hbase-options-list)]
   [SALT LIKE TABLE]

hbase-options-list is:
   hbase-option = 'value'[, hbase-option = 'value']...
```

Syntax Description of CREATE INDEX

index

is an SQL identifier that specifies the simple name for the new index. You cannot qualify *index* with its schema name. Indexes have their own namespace within a schema, so an index name might be the same as a table or constraint name. However, no two indexes in a schema can have the same name.

table

is the name of the table for which to create the index. See "Database Object Names" (page 198).

column-name [ASC[ENDING] | DESC[ENDING]] [,*column-name* [ASC[ENDING] | DESC[ENDING]]]...

specifies the columns in *table* to include in the index. The order of the columns in the index need not correspond to the order of the columns in the table.

ASCENDING or DESCENDING specifies the storage and retrieval order for rows in the index. The default is ASCENDING.

Rows are ordered by values in the first column specified for the index. If multiple index rows share the same value for the first column, the values in the second column are used to order the rows, and so forth. If duplicate index rows occur in a nonunique index, their order is based on the sequence specified for the columns of the key of the underlying table. For ordering (but not for other purposes), nulls are greater than other values.

HBASE_OPTIONS (*hbase-option* = '*value*'[, *hbase-option* = '*value*']...)

a list of HBase options to set for the index. These options are applied independently of any HBase options set for the index's table.

hbase-option = '*value*'

is one of the these HBase options and its assigned value:

HBase Option	Accepted Values[1]
BLOCKCACHE	**'true'** | 'false'
BLOCKSIZE	**'65536'** | '*positive-integer*'
BLOOMFILTER	'NONE' | **'ROW'** | 'ROWCOL'
CACHE_BLOOMS_ON_WRITE	**'true'** | 'false'
CACHE_DATA_ON_WRITE	**'true'** | 'false'
CACHE_INDEXES_ON_WRITE	**'true'** | 'false'
COMPACT	**'true'** | 'false'
COMPACT_COMPRESSION	'GZ' | 'LZ4' | 'LZO' | **'NONE'** | 'SNAPPY'
COMPRESSION	'GZ' | 'LZ4' | 'LZO' | **'NONE'** | 'SNAPPY'
DATA_BLOCK_ENCODING	'DIFF' | 'FAST_DIFF' | **'NONE'** | 'PREFIX'
DURABILITY	'USE_DEFAULT' | **'SKIP_WAL'** | 'ASYNC_WAL' | 'SYNC_WAL' | 'FSYNC_WAL'
EVICT_BLOCKS_ON_CLOSE	'true' | **'false'**
IN_MEMORY	'true' | **'false'**
KEEP_DELETED_CELLS	'true' | **'false'**
MAX_FILESIZE	'*positive-integer*'
MAX_VERSIONS	**'1'** | '*positive-integer*'
MEMSTORE_FLUSH_SIZE	'*positive-integer*'
MIN_VERSIONS	**'0'** | '*positive-integer*'
PREFIX_LENGTH_KEY	'*positive-integer*', which should be less than maximum length of the key for the table. It applies only if the SPLIT_POLICY is KeyPrefixRegionSplitPolicy.
REPLICATION_SCOPE	**'0'** | '1'
SPLIT_POLICY	**'org.apache.hadoop.hbase.regionserver.ConstantSizeRegionSplitPolicy'** | 'org.apache.hadoop.hbase.regionserver.IncreasingToUpperBoundRegionSplitPolicy' | 'org.apache.hadoop.hbase.regionserver.KeyPrefixRegionSplitPolicy'
TTL	**'-1'** (forever) | '*positive-integer*'

[1] Values in boldface are default values.

SALT LIKE TABLE

causes the index to use the same salting scheme (that is, `SALT USING num PARTITIONS [ON (column[, column]...)]`) as its base table.

Considerations for CREATE INDEX

Indexes are created under a single transaction. When an index is created, the following steps occur:

- Transaction begins (either a user-started transaction or a system-started transaction).
- Rows are written to the metadata.
- Physical labels are created to hold the index (as non audited).

- The base table is locked for read shared access which prevents inserts, updates, and deletes on the base table from occurring.
- The index is loaded by reading the base table for read uncommitted access using side tree inserts.

  NOTE:
  A side tree insert is a fast way of loading data that can perform specialized optimizations because the partitions are not audited and empty.

- After load is complete, the index audit attribute is turned on and it is attached to the base table (to bring the index online).
- The transaction is committed, either by the system or later by the requestor.

If the operation fails after basic semantic checks are performed, the index no longer exists and the entire transaction is rolled back even if it is a user-started transaction.

Authorization and Availability Requirements

An index always has the same security as the table it indexes.

CREATE INDEX locks out INSERT, DELETE, and UPDATE operations on the table being indexed. If other processes have rows in the table locked when the operation begins, CREATE INDEX waits until its lock request is granted or timeout occurs.

You cannot access an index directly.

Required Privileges

To issue a CREATE INDEX statement, one of the following must be true:

- You are DB__ROOT.
- You are creating the table in a shared schema.
- You are the private schema owner.
- You are the owner of the table.
- You have the ALTER, ALTER_TABLE, CREATE, or CREATE_INDEX component privilege for the SQL_OPERATIONS component.

  NOTE: In this case, if you create an index in a private schema, it will be owned by the schema owner.

Limits on Indexes

For nonunique indexes, the sum of the lengths of the columns in the index plus the sum of the length of the clustering key of the underlying table cannot exceed 2048 bytes.

No restriction exists on the number of indexes per table.

Examples of CREATE INDEX

- This example creates an index on two columns of a table:

```
CREATE INDEX xempname
  ON persnl.employee (last_name, first_name);
```

CREATE LIBRARY Statement

- "Syntax Description of CREATE LIBRARY"
- "Considerations for CREATE LIBRARY"
- "Examples of CREATE LIBRARY"

The CREATE LIBRARY statement registers a library object in a Trafodion database. A library object can be an SPJ's JAR file or a UDF's library file.

CREATE LIBRARY is a Trafodion SQL extension.

NOTE: DDL statements are not currently supported in transactions. That means that you cannot run this statement inside a user-defined transaction (BEGIN WORK...COMMIT WORK) or when AUTOCOMMIT is OFF. To run this statement, AUTOCOMMIT must be turned ON (the default) for the session.

```
CREATE LIBRARY [[catalog-name.]schema-name.]library-name
   FILE 'library-filename'
   [HOST NAME 'host-name']
   [LOCAL FILE 'host-filename']
```

Syntax Description of CREATE LIBRARY

[[*catalog-name*.]*schema-name*.]*library-name*

specifies the ANSI logical name of the library object, where each part of the name is a valid SQL identifier with a maximum of 128 characters. Specify a name that is unique and does not exist for libraries in the same schema. If you do not fully qualify the library name, Trafodion SQL qualifies it according to the schema of the current session. For more information, see "Identifiers" (page 221) and "Database Object Names" (page 198).

FILE '*library-filename*'

specifies the full path of a deployed library file, which either an SPJ's JAR file or a UDF's library file.

NOTE: Make sure to upload the library file to the Trafodion cluster and then copy the library file to the same directory on all the nodes in the cluster before running the CREATE LIBRARY statement. Otherwise, you will see an error message indicating that the JAR or DLL file was not found.

HOST NAME '*host-name*'

specifies the name of the client host machine where the deployed file resides.

LOCAL FILE '*host-filename*'

specifies the path on the client host machine where the deployed file is stored.

Considerations for CREATE LIBRARY

- A library object cannot refer to a library file referenced by another library object. If the *library-filename* is in use by another library object, the CREATE LIBRARY command will fail.
- The *library-filename* must specify an existing file. Otherwise, the CREATE LIBRARY command will fail.
- The CREATE LIBRARY command does not verify that the specified *library-filename* is a valid executable file.
- HOST NAME and LOCAL FILE are position dependent.

Required Privileges

To issue a CREATE LIBRARY statement, one of the following must be true:

- You are DB__ROOT.
- You are creating the library in a shared schema and have the MANAGE_LIBRARY privilege.
- You are the private schema owner and have the MANAGE_LIBRARY privilege.
- You have the CREATE or CREATE_LIBRARY component privilege for the SQL_OPERATIONS component and have the MANAGE_LIBRARY privilege.

NOTE: In this case, if you create a library in a private schema, it will be owned by the schema owner.

Examples of CREATE LIBRARY

- This CREATE LIBRARY statement registers a library named SALESLIB in the SALES schema for a JAR file (SPJs):

```
CREATE LIBRARY sales.saleslib FILE '/opt/home/trafodion/spjjars/Sales.jar';
```

- This CREATE LIBRARY statement registers a library named MYUDFS in the default schema for a library file (UDFs):

```
CREATE LIBRARY myudfs FILE $UDFLIB;
```

CREATE PROCEDURE Statement

- "Syntax Description of CREATE PROCEDURE"
- "Considerations for CREATE PROCEDURE"
- "Examples of CREATE PROCEDURE"

The CREATE PROCEDURE statement registers a Java method as a stored procedure in Java (SPJ) within a Trafodion database.

NOTE: DDL statements are not currently supported in transactions. That means that you cannot run this statement inside a user-defined transaction (BEGIN WORK...COMMIT WORK) or when AUTOCOMMIT is OFF. To run this statement, AUTOCOMMIT must be turned ON (the default) for the session.

```
CREATE PROCEDURE procedure-ref([sql-parameter-list])
   EXTERNAL NAME 'java-method-name [java-signature]'
   LIBRARY [[catalog-name.]schema-name.]library-name
   [EXTERNAL SECURITY external-security-type]
   LANGUAGE JAVA
   PARAMETER STYLE JAVA
   [NO SQL | CONTAINS SQL | MODIFIES SQL DATA | READS SQL DATA]
   [DYNAMIC RESULT SETS integer]
   [TRANSACTION REQUIRED | NO TRANSACTION REQUIRED]
   [DETERMINISTIC | NOT DETERMINISTIC]
   [NO ISOLATE | ISOLATE]

procedure-ref is:
   [[catalog-name.]schema-name.]procedure-name

sql-parameter-list is:
   sql-parameter[, sql-parameter]...

sql-parameter is:
   [parameter-mode] [sql-identifier] sql-datatype

parameter-mode is:
     IN
   | OUT
   | INOUT

java-method-name is:
   [package-name.]class-name.method-name

java-signature is:
   ([java-parameter-list])

java-parameter-list is:
   java-datatype[, java-datatype]...

external-security-type is:
     DEFINER
   | INVOKER
```

NOTE: Delimited variables in this syntax diagram are case-sensitive. Case-sensitive variables include *java-method-name*, *java-signature*, and *class-file-path*, and any delimited part of the *procedure-ref*. The remaining syntax is not case-sensitive.

Syntax Description of CREATE PROCEDURE

procedure-ref([*sql-parameter*[, *sql-parameter*]...])

specifies the name of the stored procedure in Java (SPJ) and any SQL parameters that correspond to the signature of the SPJ method.

procedure-ref

specifies an ANSI logical name of the form:

```
[[catalog-name.]schema-name.]procedure-name
```

where each part of the name is a valid SQL identifier with a maximum of 128 characters. For more information, see "Identifiers" (page 221) and "Database Object Names" (page 198).

Specify a name that is unique and does not exist for any procedure or function in the same schema. Trafodion SQL does not support the overloading of procedure names. That is, you cannot register the same procedure name more than once with different underlying SPJ methods.

If you do not fully qualify the procedure name, Trafodion SQL qualifies it according to the schema of the current session.

sql-parameter

specifies an SQL parameter that corresponds to the signature of the SPJ method:

```
[parameter-mode] [sql-identifier] sql-datatype
```

parameter-mode

specifies the mode IN, OUT, or INOUT of a parameter. The default is IN.

IN

specifies a parameter that passes data to an SPJ.

OUT

specifies a parameter that accepts data from an SPJ. The parameter must be an array.

INOUT

specifies a parameter that passes data to and accepts data from an SPJ. The parameter must be an array.

sql-identifier

specifies an SQL identifier for the parameter. For more information, see "Identifiers" (page 221).

sql-datatype

specifies an SQL data type that corresponds to the Java parameter of the SPJ method. *sql-datatype* can be:

SQL Data Type	Maps to Java Data Type...
CHAR[ACTER] CHAR[ACTER] VARYING VARCHAR PIC[TURE] X[1] NCHAR NCHAR VARYING NATIONAL CHAR[ACTER] NATIONAL CHAR[ACTER] VARYING	`java.lang.String`
DATE	`java.sql.Date`
TIME	`java.sql.Time`
TIMESTAMP	`java.sql.Timestamp`
NUMERIC (including NUMERIC with a precision greater than eighteen)[2] DEC[IMAL][2] PIC[TURE] S9[3]	`java.math.BigDecimal`

SQL Data Type	Maps to Java Data Type...
SMALLINT[2]	`short`
INT[EGER][2]	`int` or `java.lang.Integer`[4]
LARGEINT[2]	`long` or `java.lang.Long`[4]
FLOAT	`double` or `java.lang.Double`[4]
REAL	`float` or `java.lang.Float`[4]
DOUBLE PRECISION	`double` or `java.lang.Double`[4]

[1] The Trafodion database stores PIC X as a CHAR data type.

[2] Numeric data types of SQL parameters must be SIGNED, which is the default in the Trafodion database.

[3] The Trafodion database stores PIC S9 as a DECIMAL or NUMERIC data type.

[4] By default, the SQL data type maps to a Java primitive type. The SQL data type maps to a Java wrapper class only if you specify the wrapper class in the Java signature of the EXTERNAL NAME clause.

For more information, see "Data Types" (page 199).

EXTERNAL NAME '*java-method-name* [*java-signature*]'

java-method-name

specifies the case-sensitive name of the SPJ method of the form:

[*package-name*.]*class-name*.*method-name*

The Java method must exist in a Java class file, *class-name*.class, within a library registered in the database. The Java method must be defined as `public` and `static` and have a return type of `void`.

If the class file that contains the SPJ method is part of a package, you must also specify the package name. If you do not specify the package name, the CREATE PROCEDURE statement fails to register the SPJ.

java-signature

specifies the signature of the SPJ method and consists of:

([*java-datatype*[, *java-datatype*]...])

The Java signature is necessary only if you want to specify a Java wrapper class (for example, `java.lang.Integer`) instead of a Java primitive data type (for example, `int`). An SQL data type maps to a Java primitive data type by default.

The Java signature is case-sensitive and must be placed within parentheses, such as `(java.lang.Integer, java.lang.Integer)`. The signature must specify each of the parameter data types in the order they appear in the Java method definition within the class file. Each Java data type that corresponds to an OUT or INOUT parameter must be followed by empty square brackets ([ ]), such as `java.lang.Integer[]`.

java-datatype

specifies a mappable Java data type. For the mapping of the Java data types to SQL data types, see *sql-datatype*.

LIBRARY [[*catalog-name*.]*schema-name*.]*library-name*

specifies the ANSI logical name of a library containing the SPJ method. If you do not fully qualify the library name, Trafodion SQL qualifies it according to the schema of the current session.

EXTERNAL SECURITY *external-security-type*

determines the privileges, or rights, that users have when executing (or calling) the SPJ. An SPJ can have one of these types of external security:

- INVOKER determines that users can execute, or invoke, the stored procedure using the privileges of the user who invokes the stored procedure. This behavior is referred to as *invoker rights* and is the default behavior if EXTERNAL SECURITY is not specified. Invoker rights allow a user who has the execute privilege on the SPJ to call the SPJ using his or her existing privileges. In this case, the user must be granted privileges to access the underlying database objects on which the SPJ operates.

  NOTE: Granting a user privileges to the underlying database objects gives the user direct access to those database objects, which could pose a risk to more sensitive or critical data to which users should not have access. For example, an SPJ might operate on a subset of the data in an underlying database object, but that database object might contain other more sensitive or critical data to which users should not have access.

- DEFINER determines that users can execute, or invoke, the stored procedure using the privileges of the user who created the stored procedure. This behavior is referred to as *definer rights*. The advantage of definer rights is that users are allowed to manipulate data by invoking the stored procedure without having to be granted privileges to the underlying database objects. That way, users are restricted from directly accessing or manipulating more sensitive or critical data in the database. However, be careful about the users to whom you grant execute privilege on an SPJ with definer external security because those users will be able to execute the SPJ without requiring privileges to the underlying database objects.

LANGUAGE JAVA

specifies that the external user-defined routine is written in the Java language.

PARAMETER STYLE JAVA

specifies that the run-time conventions for arguments passed to the external user-defined routine are those of the Java language.

NO SQL

specifies that the SPJ cannot perform SQL operations.

CONTAINS SQL | MODIFIES SQL DATA | READS SQL DATA

specifies that the SPJ can perform SQL operations. All these options behave the same as CONTAINS SQL, meaning that the SPJ can read and modify SQL data. Use one of these options to register a method that contains SQL statements. If you do not specify an SQL access mode, the default is CONTAINS SQL.

DYNAMIC RESULT SETS *integer*

specifies the maximum number of result sets that the SPJ can return. This option is applicable only if the method signature contains a `java.sql.ResultSet[]` object. If the method contains a result set object, the valid range is 1 to 255 inclusive. The actual number of result sets returned by the SPJ method can be less than or equal to this number. If you do not specify this option, the default value is 0 (zero), meaning that the SPJ does not return result sets.

TRANSACTION REQUIRED | NO TRANSACTION REQUIRED

determines whether the SPJ must run in a transaction inherited from the calling application (TRANSACTION REQUIRED, the default option) or whether the SPJ runs without inheriting the calling application's transaction (NO TRANSACTION REQUIRED). Typically, you will want the stored procedure to inherit the transaction from the calling application. However, if the SPJ method does not access the database or if you want the stored procedure to manage its own transactions, you should set the stored procedure's transaction attribute to NO TRANSACTION REQUIRED. For more information, see "Effects of the Transaction Attribute on SPJs" (page 62).

DETERMINISTIC | NOT DETERMINISTIC

specifies whether the SPJ always returns the same values for OUT and INOUT parameters for a given set of argument values (DETERMINISTIC) or does not return the same values (NOT DETERMINISTIC, the default option). If you specify DETERMINISTIC, Trafodion SQL is not required to call the SPJ each time to produce results; instead, Trafodion SQL caches the results and reuses them during subsequent calls, thus optimizing the CALL statement.

NO ISOLATE | ISOLATE

specifies that the SPJ executes either in the environment of the database server (NO ISOLATE) or in an isolated environment (ISOLATE, the default option). Trafodion SQL allows both options but always executes the SPJ in the UDR server process (ISOLATE).

Considerations for CREATE PROCEDURE

Required Privileges

To issue a CREATE PROCEDURE statement, one of the following must be true:

- You are DB__ROOT.
- You are creating the procedure in a shared schema, and you have the USAGE (or ALL) privilege on the library that will be used in the creation of the stored procedure. The USAGE privilege provides you with read access to the library's underlying JAR file, which contains the SPJ Java method.
- You are the private schema owner and have the USAGE (or ALL) privilege on the library that will be used in the creation of the stored procedure. The USAGE privilege provides you with read access to the library's underlying JAR file, which contains the SPJ Java method.
- You have the CREATE or CREATE_ROUTINE component level privilege for the SQL_OPERATIONS component and have the USAGE (or ALL) privilege on the library that will be used in the creation of the stored procedure. The USAGE privilege provides you with read access to the library's underlying JAR file, which contains the SPJ Java method.

NOTE: In this case, if you create a stored procedure in a private schema, it will be owned by the schema owner.

Effects of the Transaction Attribute on SPJs

Transaction Required

Using Transaction Control Statements or Methods

If you specify TRANSACTION REQUIRED (the default option), a CALL statement automatically initiates a transaction if there is no active transaction. In this case, you should not use transaction control statements (or equivalent JDBC transaction methods) in the SPJ method. Transaction control statements include COMMIT WORK and ROLLBACK WORK, and the equivalent JDBC transaction methods are `Connection.commit()` and `Connection.rollback()`. If you try to use transaction control statements or methods in an SPJ method when the stored procedure's transaction attribute is set to TRANSACTION REQUIRED, the transaction control statements or methods in the SPJ method are ignored, and the Java virtual machine (JVM) does not report any errors or warnings. When the stored procedure's transaction attribute is set to TRANSACTION REQUIRED, you should rely on the transaction control statements or methods in the application that calls the stored procedure and allow the calling application to manage the transactions.

Committing or Rolling Back a Transaction

If you do not use transaction control statements in the calling application, the transaction initiated by the CALL statement might not automatically commit or roll back changes to the database. When AUTOCOMMIT is ON (the default setting), the database engine automatically commits or rolls

back any changes made to the database at the end of the CALL statement execution. However, when AUTOCOMMIT is OFF, the current transaction remains active until the end of the client session or until you explicitly commit or roll back the transaction. To ensure an atomic unit of work when calling an SPJ, use the COMMIT WORK statement in the calling application to commit the transaction when the CALL statement succeeds, and use the ROLLBACK WORK statement to roll back the transaction when the CALL statement fails.

No Transaction Required

In some cases, you might not want the SPJ method to inherit the transaction from the calling application. Instead, you might want the stored procedure to manage its own transactions or to run without a transaction. Not inheriting the calling application's transaction is useful in these cases:

- The stored procedure performs several long-running operations, such as multiple DDL or table maintenance operations, on the database. In this case, you might want to commit those operations periodically from within the SPJ method to avoid locking tables for a long time.
- The stored procedure performs certain SQL operations that must run without an active transaction. For example, INSERT, UPDATE, and DELETE statements with the WITH NO ROLLBACK option are rejected when a transaction is already active, as is the case when a stored procedure inherits a transaction from the calling application. The PURGEDATA utility is also rejected when a transaction is already active.
- The stored procedure does not access the database. In this case, the stored procedure does not need to inherit the transaction from the calling application. By setting the stored procedure's transaction attribute to NO TRANSACTION REQUIRED, you can avoid the overhead of the calling application's transaction being propagated to the stored procedure.

In these cases, you should set the stored procedure's transaction attribute to NO TRANSACTION REQUIRED when creating the stored procedure.

If you specify NO TRANSACTION REQUIRED and if the SPJ method creates a JDBC default connection, that connection will have autocommit enabled by default. You can either use the autocommit transactions or disable autocommit (`conn.setAutoCommit(false);`) and use the JDBC transaction methods, `Connection.commit()` and `Connection.rollback()`, to commit or roll back work where needed.

Examples of CREATE PROCEDURE

- This CREATE PROCEDURE statement registers an SPJ named LOWERPRICE, which does not accept any arguments:

```
SET SCHEMA SALES;

CREATE PROCEDURE lowerprice()
  EXTERNAL NAME 'Sales.lowerPrice'
  LIBRARY saleslib
  LANGUAGE JAVA
  PARAMETER STYLE JAVA
  MODIFIES SQL DATA;
```

  Because the procedure name is not qualified by a catalog and schema, Trafodion SQL qualifies it according to the current session settings, where the catalog is TRAFODION (by default) and the schema is set to SALES. Since the procedure needs to be able to read and modify SQL data, MODIFIES SQL DATA is specified in the CREATE PROCEDURE statement.

  To call this SPJ, use this CALL statement:

```
CALL lowerprice();
```

The LOWERPRICE procedure lowers the price of items with 50 or fewer orders by 10 percent in the database.

- This CREATE PROCEDURE statement registers an SPJ named TOTALPRICE, which accepts three input parameters and returns a numeric value, the total price to an INOUT parameter:

```
CREATE PROCEDURE trafodion.sales.totalprice(IN qty NUMERIC (18),
                                            IN rate VARCHAR (10),
                                            INOUT price NUMERIC (18,2))
  EXTERNAL NAME 'Sales.totalPrice'
  LIBRARY sales.saleslib
  LANGUAGE JAVA
  PARAMETER STYLE JAVA
  NO SQL;
```

To call this SPJ in TrafCI, use these statements:

```
SET PARAM ?p 10.00;

CALL sales.totalprice(23, 'standard', ?p);

p
--------------------
              253.97

--- SQL operation complete.
```

Since the procedure does not read and modify any SQL data, NO SQL is specified in the CREATE PROCEDURE statement.

- This CREATE PROCEDURE statement registers an SPJ named MONTHLYORDERS, which accepts an integer value for the month and returns the number of orders:

```
CREATE PROCEDURE sales.monthlyorders(IN INT, OUT number INT)
  EXTERNAL NAME 'Sales.numMonthlyOrders (int, java.lang.Integer[])'
  LIBRARY sales.saleslib
  LANGUAGE JAVA
  PARAMETER STYLE JAVA
  READS SQL DATA;
```

Because the OUT parameter is supposed to map to the Java wrapper class, `java.lang.Integer`, you must specify the Java signature in the EXTERNAL NAME clause.

To invoke this SPJ, use this CALL statement:

```
CALL sales.monthlyorders(3, ?);

ORDERNUM
-----------
          4

--- SQL operation complete.
```

- This CREATE PROCEDURE statement registers an SPJ named ORDERSUMMARY, which accepts a date (formatted as a string) and returns information about the orders on or after that date.

```
CREATE PROCEDURE sales.ordersummary(IN on_or_after_date VARCHAR (20),
                                    OUT num_orders LARGEINT)
  EXTERNAL NAME 'Sales.orderSummary (int, long[])'
  LIBRARY sales.saleslib
  EXTERNAL SECURITY invoker
  LANGUAGE JAVA
  PARAMETER STYLE JAVA
  READS SQL DATA
  DYNAMIC RESULT SETS 2;
```

To invoke this SPJ, use this CALL statement:

```
CALL neo.sales.ordersummary('01-01-2014', ?);
```

The ORDERSUMMARY procedure returns this information about the orders on or after the specified date, 01-01-2014:

```
NUM_ORDERS
--------------------
                  13

ORDERNUM NUM_PARTS             AMOUNT               ORDER_DATE LAST_NAME
-------- -------------------- -------------------- ---------- --------------------
  100210                    4 19020.00             2014-04-10 HUGHES
  100250                    4 22625.00             2014-01-23 HUGHES
  101220                    4 45525.00             2014-07-21 SCHNABL
  ... ... ... ... ...

--- 13 row(s) selected.

ORDERNUM PARTNUM UNIT_PRICE QTY_ORDERED PARTDESC
-------- ------- ---------- ----------- ------------------
  100210     244    3500.00           3 PC GOLD, 30 MB
  100210    2001    1100.00           3 GRAPHIC PRINTER,M1
  100210    2403     620.00           6 DAISY PRINTER,T2
  ... ... ... ... ...

--- 70 row(s) selected.

--- SQL operation complete.
```

CREATE ROLE Statement

- "Syntax Description of CREATE ROLE"
- "Considerations for CREATE ROLE"
- "Examples of CREATE ROLE"

The CREATE ROLE statement creates an SQL role. See "Roles" (page 248).

```
CREATE ROLE role-name [ WITH ADMIN grantor ]

grantor is:
     database-username
```

Syntax Description of CREATE ROLE

role-name
: is an SQL identifier that specifies the new role. *role-name* is a regular or delimited case-insensitive identifier. See "Case-Insensitive Delimited Identifiers" (page 221). *role-name* cannot be an existing role name, and it cannot be a registered database username. However, *role-name* can be a configured directory-service username.

WITH ADMIN *grantor*
: specifies a role owner other than the current user. This is an optional clause.

grantor
: specifies a registered database username to whom you assign the role owner.

Considerations for CREATE ROLE

- To create a role, you must either be DB__ROOT or have been granted the MANAGE_ROLES component privilege for SQL_OPERATIONS.
- PUBLIC, _SYSTEM, NONE, and database usernames beginning with DB__ are reserved. You cannot specify a *role-name* with any such name.

Role Ownership

You can give role ownership to a user by specifying the user in the WITH ADMIN *grantor* clause with the *grantor* as the user.

The role owner can perform these operations:

- Grant and revoke the role to users.
- Drop the role.

Role ownership is permanent. After you create the role, the ownership of the role cannot be changed or assigned to another user.

Examples of CREATE ROLE

- To create a role and assign the current user as the role owner:

  ```
  CREATE ROLE clerks;
  ```

- To create a role and assign another user as the role owner:

  ```
  CREATE ROLE sales WITH ADMIN cmiller;
  ```

CREATE SCHEMA Statement

- "Syntax Description of CREATE SCHEMA"
- "Considerations for CREATE SCHEMA"
- "Examples of CREATE SCHEMA"

The CREATE SCHEMA statement creates a schema in the database. See "Schemas" (page 249).

NOTE: DDL statements are not currently supported in transactions. That means that you cannot run DDL statements inside a user-defined transaction (BEGIN WORK...COMMIT WORK) or when AUTOCOMMIT is OFF. To run these statements, AUTOCOMMIT must be turned ON (the default) for the session.

```
CREATE [schema-class] SCHEMA schema-clause

schema-class is:
  [ PRIVATE | SHARED ]

schema-clause is:
  { schema-name [AUTHORIZATION authid] | AUTHORIZATION authid }
```

Syntax Description of CREATE SCHEMA

schema-class

indicates whether access to the schema is restricted to the authorization ID by default (PRIVATE) or whether any database user may add objects to the schema (SHARED). The default class is PRIVATE.

NOTE: Schemas created in Trafodion Release 0.9 or earlier are SHARED schemas.

schema-name

is a name for the new schema and is an SQL identifier that specifies a unique name that is not currently a schema name. This parameter is optional. However, if you do not specify a schema name, you must specify the authorization clause. If a schema name is not provided, the authorization ID is used for the schema name. If the authorization ID name matches an existing schema, the CREATE SCHEMA command fails.

authid

is the name of the database user or role will own and administer the schema. If this clause is not present, the current user becomes the schema owner.

Considerations for CREATE SCHEMA

Reserved Schema Names

Schema names that begin with a leading underscore (_) are reserved for future use.

AUTHORIZATION Clause

The AUTHORIZATION clause is optional. If you omit this clause, the current user becomes the schema owner.

NOTE: An authorization ID is assigned to a schema name even if authorization is not enabled for the Trafodion database. However, no enforcement occurs unless authorization is enabled.

The schema owner can perform operations on the schema and on objects within the schema. For example:

- Alter DDL of objects
- Drop the schema
- Drop objects
- Manage objects with utility commands such as UPDATE STATISTICS and PURGEDATA

Who Can Create a Schema

The privilege to create a schema is controlled by the component privilege CREATE_SCHEMA for the SQL_OPERATIONS component. By default, this privilege is granted to PUBLIC, but it can be revoked by DB__ROOT.

When authorization is initialized, these authorization IDs are granted the CREATE_SCHEMA privilege:

- PUBLIC
- DB__ROOT
- DB__ROOTROLE

DB__ROOT or anyone granted the DB_ROOTROLE role can grant the CREATE_SCHEMA privilege.

Examples of CREATE SCHEMA

- This example creates a private schema schema named MYSCHEMA, which will be owned by the current user:

  ```
  CREATE SCHEMA myschema;
  ```

- This example creates a shared schema and designates `CliffG` as the schema owner:

  ```
  CREATE SHARED SCHEMA hockey_league AUTHORIZATION "CliffG";
  ```

- This example creates a private schema and designates the role DBA as the schema owner:

  ```
  CREATE PRIVATE SCHEMA contracts AUTHORIZATION DBA;
  ```

  Users with the role DBA granted to them can grant access to objects in the CONTRACTS schema to other users and roles.

- This example creates a schema named JSMITH:

  ```
  CREATE PRIVATE SCHEMA AUTHORIZATION JSmith;
  ```

CREATE TABLE Statement

- "Syntax Description of CREATE TABLE"
- "Considerations for CREATE TABLE"
- "Examples of CREATE TABLE"

The CREATE TABLE statement creates a Trafodion SQL table, which is a mapping of a relational SQL table to an HBase table. The CREATE VOLATILE TABLE statement creates a temporary Trafodion SQL table that exists only during an SQL session. The CREATE TABLE AS statement creates a table based on the data attributes of a SELECT query and populates the table using the data returned by the SELECT query. See "Tables" (page 254).

NOTE: DDL statements are not currently supported in transactions. That means that you cannot run this statement inside a user-defined transaction (BEGIN WORK...COMMIT WORK) or when AUTOCOMMIT is OFF. To run this statement, AUTOCOMMIT must be turned ON (the default) for the session.

```
CREATE [VOLATILE] TABLE IF NOT EXISTS table
    { table-spec | like-spec }
    [SALT USING num PARTITIONS [ON (column[, column]...)]]
    [STORE BY {PRIMARY KEY | (key-column-list)}]
    [HBASE_OPTIONS (hbase-options-list)]
    [LOAD IF EXISTS | NO LOAD]
    [AS select-query]

table-spec is:
   (table-element [,table-element]...)

table-element is:
     column-definition
   | [CONSTRAINT constraint-name] table-constraint

column-definition is:
   column data-type
   [DEFAULT default | NO DEFAULT]
   [[CONSTRAINT constraint-name] column-constraint]...

data-type is:
   CHAR[ACTER] [(length [CHARACTERS])]
         [CHARACTER SET char-set-name]
         [UPSHIFT] [[NOT]CASESPECIFIC]
   | CHAR[ACTER] VARYING (length [CHARACTERS])
         [CHARACTER SET char-set-name]
         [UPSHIFT] [[NOT]CASESPECIFIC]
   | VARCHAR (length) [CHARACTER SET char-set-name]
         [UPSHIFT] [[NOT]CASESPECIFIC]
   | NCHAR (length) [CHARACTERS] [UPSHIFT] [[NOT]CASESPECIFIC]
   | NCHAR VARYING(length [CHARACTERS]) [UPSHIFT] [[NOT] CASESPECIFIC]
   | NUMERIC [(precision [,scale])] [SIGNED|UNSIGNED]
   | SMALLINT [SIGNED|UNSIGNED]
   | INT[EGER] [SIGNED|UNSIGNED]
   | LARGEINT
   | DEC[IMAL] [(precision [,scale])] [SIGNED|UNSIGNED]
   | FLOAT [(precision)]
   | REAL
   | DOUBLE PRECISION
   | DATE
   | TIME [(time-precision)]
   | TIMESTAMP [(timestamp-precision)]
   | INTERVAL { start-field TO end-field | single-field }
```

```
default is:
     literal
   | NULL
   | CURRENT_DATE
   | CURRENT_TIME
   | CURRENT_TIMESTAMP

column-constraint is:
     NOT NULL
   | UNIQUE
   | PRIMARY KEY [ASC[ENDING] | DESC[ENDING]]
   | CHECK (condition)
   | REFERENCES ref-spec

table-constraint is:
     UNIQUE (column-list)
   | PRIMARY KEY (key-column-list)
   | CHECK (condition)
   | FOREIGN KEY (column-list) REFERENCES ref-spec

ref-spec is:
   referenced-table [(column-list)]

column-list is:
   column-name [,column-name]...

key-column-list is:
   column-name [ASC[ENDING] | DESC[ENDING]]
      [,column-name [ASC[ENDING] | DESC[ENDING]]]...

like-spec is:
   LIKE source-table [include-option]

hbase-options-list is:
   hbase-option = 'value'[, hbase-option = 'value']...
```

Syntax Description of CREATE TABLE

VOLATILE

specifies a volatile table, which is a table limited to the session that creates the table. After the session ends, the table is automatically dropped. See "Considerations for CREATE VOLATILE TABLE" (page 75).

IF NOT EXISTS

creates an HBase table if it does not already exist when the table is created. This option does not apply to volatile tables.

table

specifies the ANSI logical name of the table. See "Database Object Names" (page 198). This name must be unique among names of tables and views within its schema.

SALT USING *num* PARTITIONS [ON (*column*[, *column*]...)]

pre-splits the table into multiple regions when the table is created. Salting adds a hash value of the row key as a key prefix, thus avoiding hot spots for sequential keys. The number of partitions that you specify can be a function of the number of region servers present in the HBase cluster. You can specify a number from 2 to 1024. If you do not specify columns, the default is to use all primary key columns.

STORE BY { PRIMARY KEY | (*key-column-list*)}

specifies a set of columns on which to base the clustering key. The clustering key determines the order of rows within the physical file that holds the table. The storage order has an effect on how you can partition the object.

PRIMARY KEY

bases the clustering key on the primary key columns.

key-column-list

bases the clustering key on the columns in the *key-column-list*. The key columns in *key-column-list* must be specified as NOT NULL and must be the same as the primary key columns that are defined on the table. If STORE BY is not specified, then the clustering key is the PRIMARY KEY.

HBASE_OPTIONS (*hbase-option* = '*value*'[, *hbase-option* = '*value*']...)

a list of HBase options to set for the table.

hbase-option = '*value*'

is one of the these HBase options and its assigned value:

HBase Option	Accepted Values[1]
BLOCKCACHE	**'true'** | 'false'
BLOCKSIZE	**'65536'** | '*positive-integer*'
BLOOMFILTER	'NONE' | **'ROW'** | 'ROWCOL'
CACHE_BLOOMS_ON_WRITE	**'true'** | 'false'
CACHE_DATA_ON_WRITE	**'true'** | 'false'
CACHE_INDEXES_ON_WRITE	**'true'** | 'false'
COMPACT	**'true'** | 'false'
COMPACT_COMPRESSION	'GZ' | 'LZ4' | 'LZO' | **'NONE'** | 'SNAPPY'
COMPRESSION	'GZ' | 'LZ4' | 'LZO' | **'NONE'** | 'SNAPPY'
DATA_BLOCK_ENCODING	'DIFF' | 'FAST_DIFF' | **'NONE'** | 'PREFIX'
DURABILITY	'USE_DEFAULT' | **'SKIP_WAL'** | 'ASYNC_WAL' | 'SYNC_WAL' | 'FSYNC_WAL'
EVICT_BLOCKS_ON_CLOSE	'true' | **'false'**
IN_MEMORY	'true' | **'false'**
KEEP_DELETED_CELLS	'true' | **'false'**
MAX_FILESIZE	'*positive-integer*'
MAX_VERSIONS	**'1'** | '*positive-integer*'
MEMSTORE_FLUSH_SIZE	'*positive-integer*'
MIN_VERSIONS	**'0'** | '*positive-integer*'
PREFIX_LENGTH_KEY	'*positive-integer*', which should be less than maximum length of the key for the table. It applies only if the SPLIT_POLICY is KeyPrefixRegionSplitPolicy.
REPLICATION_SCOPE	**'0'** | '1'
SPLIT_POLICY	**'org.apache.hadoop.hbase.regionserver.ConstantSizeRegionSplitPolicy'** | 'org.apache.hadoop.hbase.regionserver.IncreasingToUpperBoundRegionSplitPolicy' | 'org.apache.hadoop.hbase.regionserver.KeyPrefixRegionSplitPolicy'
TTL	**'-1'** (forever) | '*positive-integer*'

[1] Values in boldface are default values.

LOAD IF EXISTS

loads data into an existing table. Must be used with AS *select-query*. See "Considerations for LOAD IF EXISTS and NO LOAD options of CREATE TABLE AS" (page 78).

NO LOAD

creates a table with the CREATE TABLE AS statement, but does not load data into the table. See "Considerations for LOAD IF EXISTS and NO LOAD options of CREATE TABLE AS" (page 78).

AS *select-query*

specifies a select query which is used to populate the created table. A select query can be any SQL select statement.

column data-type

specifies the name and data type for a column in the table. At least one column definition is required in a CREATE TABLE statement.

column is an SQL identifier. *column* must be unique among column names in the table. If the name is a Trafodion SQL reserved word, you must delimit it by enclosing it in double quotes. Such delimited parts are case-sensitive. For example: `"join"`.

data-type is the data type of the values that can be stored in *column*. A default value must be of the same type as the column, including the character set for a character column. See "Data Types" (page 199). Data type also includes case specific information, such as UPSHIFT.

[NOT] CASESPECIFIC

specifies that the column contains strings that are not case specific. The default is CASESPECIFIC. Comparison between two values is done in a case insensitive way only if both are case insensitive. This applies to comparison in a binary predicate, LIKE predicate, and POSITION/REPLACE string function searches. See "Examples of CREATE TABLE" (page 79).

DEFAULT *default* | NO DEFAULT

specifies a default value for the column or specifies that the column does not have a default value. "DEFAULT Clause" (page 257).

CONSTRAINT *constraint-name*

specifies a name for the column or table constraint. *constraint-name* must have the same schema as *table* and must be unique among constraint names in its schema. If you omit the schema portions of the name you specify in *constraint-name*, Trafodion SQL expands the constraint name by using the schema for *table*. See "Constraint Names" (page 195) and "Database Object Names" (page 198).

NOT NULL

is a column constraint that specifies that the column cannot contain nulls. If you omit NOT NULL, nulls are allowed in the column. If you specify both NOT NULL and NO DEFAULT, each row inserted in the table must include a value for the column. See "Null" (page 231).

UNIQUE, or, UNIQUE (*column-list*)

is a column or table constraint, respectively, that specifies that the column or set of columns cannot contain more than one occurrence of the same value or set of values. If you omit UNIQUE, duplicate values are allowed unless the column is part of the PRIMARY KEY.

column-list cannot include more than one occurrence of the same column. In addition, the set of columns that you specify on a UNIQUE constraint cannot match the set of columns on any other UNIQUE constraint for the table or on the PRIMARY KEY constraint for the table. All columns defined as unique must be specified as NOT NULL.

A UNIQUE constraint is enforced with a unique index. If there is already a unique index on *column-list*, Trafodion SQL uses that index. If a unique index does not exist, the system creates a unique index.

PRIMARY KEY [ASC[ENDING] | DESC[ENDING]], or, PRIMARY KEY (*key-column-list*)

is a column or table constraint, respectively, that specifies a column or set of columns as the primary key for the table. *key-column-list* cannot include more than one occurrence of the same column.

ASCENDING and DESCENDING specify the direction for entries in one column within the key. The default is ASCENDING.

The PRIMARY KEY value in each row of the table must be unique within the table. A PRIMARY KEY defined for a set of columns implies that the column values are unique and not null. You can specify PRIMARY KEY only once on any CREATE TABLE statement.

Trafodion SQL uses the primary key as the clustering key of the table to avoid creating a separate, unique index to implement the primary key constraint.

A PRIMARY KEY constraint is required in Trafodion SQL.

CHECK (*condition*)

is a constraint that specifies a condition that must be satisfied for each row in the table. See "Search Condition" (page 250).

You cannot refer to the CURRENT_DATE, CURRENT_TIME, or CURRENT_TIMESTAMP function in a CHECK constraint, and you cannot use subqueries in a CHECK constraint.

REFERENCES *ref-spec*

specifies a REFERENCES column constraint. The maximum combined length of the columns for a REFERENCES constraint is 2048 bytes.

ref-spec is:

referenced-table [(*column-list*)]

referenced-table is the table referenced by the foreign key in a referential constraint. *referenced-table* cannot be a view. *referenced-table* cannot be the same as *table*. *referenced-table* corresponds to the foreign key in the *table*.

column-list specifies the column or set of columns in the *referenced-table* that corresponds to the foreign key in *table*. The columns in the column list associated with REFERENCES must be in the same order as the columns in the column list associated with FOREIGN KEY. If *column-list* is omitted, the referenced table's PRIMARY KEY columns are the referenced columns.

A table can have an unlimited number of referential constraints, and you can specify the same foreign key in more than one referential constraint, but you must define each referential constraint separately. You cannot create self-referencing foreign key constraints.

FOREIGN KEY (*column-list*) REFERENCES *ref-spec*

is a table constraint that specifies a referential constraint for the table, declaring that a column or set of columns (called a foreign key) in *table* can contain only values that match those in a column or set of columns in the table specified in the REFERENCES clause.

The two columns or sets of columns must have the same characteristics (data type, length, scale, precision). Without the FOREIGN KEY clause, the foreign key in *table* is the column being defined; with the FOREIGN KEY clause, the foreign key is the column or set of columns specified in the FOREIGN KEY clause. For information about *ref-spec*, see REFERENCES *ref-spec*.

LIKE *source-table* [*include-option*]...

directs Trafodion SQL to create a table like the existing table, *source-table*, omitting constraints (with the exception of the NOT NULL and PRIMARY KEY constraints) and partitions unless the *include-option* clauses are specified.

source-table

is the ANSI logical name for the existing table and must be unique among names of tables and views within its schema.

include-option

WITH CONSTRAINTS

directs Trafodion SQL to use constraints from *source-table*. Constraint names for *table* are randomly generated unique names.

When you perform a CREATE TABLE LIKE, whether or not you include the `WITH CONSTRAINTS` clause, the target table will have all the NOT NULL column constraints that exist for the source table with different constraint names.

WITH PARTITIONS

directs Trafodion SQL to use partition definitions from *`source-table`*. Each new table partition resides on the same volume as its original *`source-table`* counterpart. The new table partitions do not inherit partition names from the original table. Instead, Trafodion SQL generates new names based on the physical file location.

If you specify the LIKE clause and the SALT USING *`num`* PARITIONS clause, you cannot specify `WITH PARTITIONS`.

Considerations for CREATE TABLE

The following subsections provide considerations for various CREATE TABLE options:

- "Authorization and Availability Requirements" (page 74)
- "Considerations for CREATE VOLATILE TABLE" (page 75)
- "Considerations for CREATE TABLE ... LIKE" (page 77)
- "Considerations for LOAD IF EXISTS and NO LOAD options of CREATE TABLE AS" (page 78)
- "Considerations for CREATE TABLE AS" (page 78)

Authorization and Availability Requirements

Required Privileges

To issue a CREATE TABLE statement, one of the following must be true:

- You are DB__ROOT.
- You are creating the table in a shared schema.
- You are the private schema owner.
- You have the CREATE or CREATE_TABLE component privilege for the SQL_OPERATIONS component.

NOTE: In this case, if you create a table in a private schema, it will be owned by the schema owner.

Privileges Needed to Create a Referential Integrity Constraint

To create a referential integrity constraint (that is, a constraint on the table that refers to a column in another table), one of the following must be true:

- You are DB__ROOT.
- You are the owner of the referencing and referenced tables.
- You have these privileges on the referencing and referenced table:
  - For the referencing table, you have the CREATE or CREATE_TABLE component privilege for the SQL_OPERATIONS component.
  - For the referenced table, you have the REFERENCES (or ALL) privilege on the referenced table through your username or through a granted role.

If the constraint refers to the other table in a query expression, you must also have SELECT privileges on the other table.

Considerations for CREATE VOLATILE TABLE

- Volatile temporary tables are closely linked to the session. Their namespace is unique across multiple concurrent sessions, and therefore allow multiple sessions to use the same volatile temporary table names simultaneously without any conflicts.
- Volatile tables support creation of indexes.
- Volatile tables are partitioned by the system. The number of partitions is limited to four partitions by default. The partitions will be distributed across the cluster. The default value is four partitions regardless of the system configuration.
- Statistics are not automatically updated for volatile tables. If you need statistics, you must explicitly run UPDATE STATISTICS.
- Volatile tables can be created and accessed using one-part, two-part, or three-part names. However, you must use the same name (one part, two part, or three part) for any further DDL or DML statements on the created volatile table. See "Examples of CREATE TABLE" (page 79).
- Trafodion SQL allows users to explicitly specify primary key and STORE BY clauses on columns that contain null values.
- Trafodion SQL does not require that the first column in a volatile table contain not null values and be the primary key. Instead, Trafodion SQL attempts to partition the table, if possible, using an appropriate suitable key column as the primary and partitioning key. For more information, see "How Trafodion SQL Selects Suitable Keys for Volatile Tables" (page 75).

Restrictions for CREATE VOLATILE TABLE

These items are not supported for volatile tables:

- ALTER statement
- User constraints
- Creating views
- Creating non-volatile indexes on a volatile table or a volatile index on a non-volatile table
- CREATE TABLE LIKE operations

How Trafodion SQL Supports Nullable Keys for Volatile Tables

- Allows nullable keys in primary key, STORE BY, and unique constraints.
- A null value is treated as the highest value for that column.
- A null value as equal to other null values and only one value is allowed for that column.

How Trafodion SQL Selects Suitable Keys for Volatile Tables

Trafodion SQL searches for the first suitable column in the list of columns of the table being created. Once the column is located, the table is partitioned on it. The searched columns in the table might be explicitly specified (as in a CREATE TABLE statement) or implicitly created (as in a CREATE TABLE AS SELECT statement).

The suitable key column is selected only if no primary key or STORE BY clause has been specified in the statement. If any of these clauses have been specified, they are used to select the key columns.

Trafodion SQL follows these guidelines to search for and select suitable keys:

- A suitable column can be a nullable column.
- Certain data types in Trafodion SQL cannot be used as a partitioning key. Currently, this includes any floating point columns (REAL, DOUBLE PRECISION, and FLOAT).

- Trafodion SQL searches for a suitable column according to this predefined order:
  - Numeric columns are chosen first, followed by fixed CHAR, DATETIME, INTERVAL, and VARCHAR data types.
  - Within numeric data types, the order is binary NUMERIC (LARGEINT, INTEGER, SMALLINT), and DECIMAL.
  - An unsigned column is given preference over a signed column.
  - A non-nullable column is given preference over a nullable column.
  - If all data types are the same, the first column is selected.
- If a suitable column is not located, the volatile table becomes a non-partitioned table with a system-defined SYSKEY as its primary key.
- If a suitable column is located, it becomes the partitioning key where the primary key is *suitable_column*, SYSKEY. This causes the table to be partitioned while preventing the duplicate key and null-to-non-null errors.

Table 1 shows the order of precedence, from low to high, of data types when Trafodion SQL searches for a suitable key. A data type appearing later has precedence over previously-appearing data types. Data types that do not appear in Table 1 cannot be chosen as a key column.

Table 1 Precedence of Data Types During Suitable Key Searches

Precedence of Data Types (From Low to High)
VARCHAR
INTERVAL
DATETIME
CHAR(ACTER)
DECIMAL (signed, unsigned)
SMALLINT (signed, unsigned)
INTEGER (signed, unsigned)
LARGEINT (signed only)

Creating Nullable Constraints in a Volatile Table

These examples show the creation of nullable constraints (primary key, STORE BY, and unique) in a volatile table:

```
create volatile table t (a int, primary key(a));
create volatile table t (a int, store by primary key);
create volatile table t (a int unique);
```

Creating a Volatile Table With a Nullable Primary Key

This example creates a volatile table with a nullable primary key:

```
>>create volatile table t (a int, primary key(a));

  --- SQL operation complete.
```

Only one unique null value is allowed:

```
>>insert into t values (null);

  --- 1 row(s) inserted.
  >>insert into t values (null);
```

```
*** ERROR[8102] The operation is prevented by a unique constraint.

--- 0 row(s) inserted.
```

Examples for Selecting Suitable Keys for Volatile Tables

These examples show the order by which Trafodion SQL selects a suitable key based on the precedence rules described in "How Trafodion SQL Selects Suitable Keys for Volatile Tables" (page 75):

- Selects column `a` as the primary and partitioning key:

  ```
  create volatile table t (a int);
  ```

- Selects column `b` because `int` has a higher precedence than `char`:

  ```
  create volatile table t (a char(10), b int);
  ```

- Selects column `b` because `not null` has precedence over `nullable` columns:

  ```
  create volatile table t (a int, b int not null);
  ```

- Selects column `b` because `int` has precedence over `decimal`:

  ```
  create volatile table t (a decimal(10), b int);
  ```

- Selects the first column, `a`, because both columns have the same data type:

  ```
  create volatile table t (a int not null, b int not null);
  ```

- Selects column `b` because `char` has precedence over `date`:

  ```
  create volatile table t (a date, b char(10));
  ```

- Selects column `b` because the `real` data type is not part of the columns to be examined:

  ```
  create volatile table t (a real, b date);
  ```

- Does not select any column as the primary/partitioning key. SYSKEY is used automatically.

  ```
  create volatile table t (a real, b double precision not null);
  ```

Similar examples would be used for CREATE TABLE AS SELECT queries.

Considerations for CREATE TABLE ... LIKE

The CREATE TABLE LIKE statement does not create views, owner information, or privileges for the new table based on the source table. Privileges associated with a new table created by using the LIKE specification are defined as if the new table is created explicitly by the current user.

CREATE TABLE ... LIKE and File Attributes

CREATE TABLE ... LIKE creates a table like another table, with the exception of file attributes. File attributes include COMPRESSION, and so on. If you do not include the attribute value as part of the CREATE TABLE ... LIKE command, SQL creates the table with the default value for the attributes and not the value from the source object. For example, to create a table like another table that specifies compression, you must specify the compression attribute value as part of the CREATE TABLE... LIKE statement. In the following example, the original CREATE TABLE statement creates a table without compression. However, in the CREATE TABLE ... LIKE statement, compression is specified.

```
-- Original Table
create table NPTEST
(FIRST_NAME CHAR(12) CHARACTER SET ISO88591 COLLATE DEFAULT NO DEFAULT
NOT NULL
, LAST_NAME CHAR(24) CHARACTER SET ISO88591 COLLATE
DEFAULT NO DEFAULT NOT NULL
  , ADDRESS CHAR(128) CHARACTER SET ISO88591 COLLATE
DEFAULT DEFAULT NULL
```

```
  , ZIP INT DEFAULT 0
  , PHONE CHAR(10) CHARACTER SET ISO88591 COLLATE
DEFAULT DEFAULT NULL , SSN LARGEINT NO DEFAULT NOT NULL
  , INFO1 CHAR(128) CHARACTER SET ISO88591 COLLATE
DEFAULT DEFAULT NULL , INFO2 CHAR(128) CHARACTER SET ISO88591 COLLATE
DEFAULT DEFAULT NULL , primary key (SSN,first_name,last_name)
)
max table size  512

-- CREATE TABLE LIKE

create table LSCE002 like NPTEST  ATTRIBUTE compression type hardware;
```

Considerations for CREATE TABLE AS

These considerations apply to CREATE TABLE AS:

- Access to the table built by CREATE TABLE AS will be a full table scan because a primary and clustering key cannot be easily defined.
- Compile time estimates and runtime information is not generated for CREATE TABLE AS tables.
- You cannot manage CREATE TABLE AS tables using WMS compile time or runtime rules.
- You cannot specify a primary key for a CREATE TABLE AS table without explicitly defining all the columns in the CREATE TABLE statement.
- You cannot generate an explain plan for a CREATE TABLE AS ...INSERT/SELECT statement. You can, however, use the EXPLAIN plan for a CREATE TABLE AS ... INSERT/SELECT statement if you use the NO LOAD option.
- You cannot use the ORDER BY clause in a CREATE TABLE AS statement. The compiler transparently orders the selected rows to improve the efficiency of the insert.

Considerations for LOAD IF EXISTS and NO LOAD options of CREATE TABLE AS

The LOAD IF EXISTS option in a CREATE TABLE AS statement causes data to be loaded into an existing table. If you do not specify the LOAD IF EXISTS option and try to load data into an existing table, the CREATE TABLE AS statement fails to execute. Use the LOAD IF EXISTS option with the AS clause in these scenarios:

- Running CREATE TABLE AS without re-creating the table. The table must be empty. Otherwise, the CREATE TABLE AS statement returns an error. Delete the data in the table by using a DELETE statement before issuing the CREATE TABLE AS statement.
- Using CREATE TABLE AS to incrementally add data to an existing table. You must start a user-defined transaction before issuing the CREATE TABLE AS statement. If you try to execute the CREATE TABLE AS statement without starting a user-defined transaction, an error is returned, stating that data already exists in the table. With a user-defined transaction, newly added rows are rolled back if an error occurs.

The NO LOAD option in a CREATE TABLE AS statement creates a table with the CREATE TABLE AS statement, but does not load data into the table. The option is useful if you must create a table to review its structure and to analyze the SELECT part of the CREATE TABLE AS statement with the EXPLAIN statement. You can also use EXPLAIN to analyze the implicated INSERT/SELECT part of the CREATE TABLE AS ... NO LOAD statement. For example:

```
CREATE TABLE ttgt NO LOAD AS (SELECT ...);
```

Trafodion SQL Extensions to CREATE TABLE

This statement is supported for compliance with ANSI SQL:1999 Entry Level. Trafodion SQL extensions to the CREATE TABLE statement are ASCENDING, DESCENDING, and PARTITION clauses. CREATE TABLE LIKE is also an extension.

Examples of CREATE TABLE

- This example creates a table. The clustering key is the primary key.

```
CREATE TABLE SALES.ODETAIL
  ( ordernum     NUMERIC (6) UNSIGNED  NO DEFAULT  NOT NULL,
    partnum      NUMERIC (4) UNSIGNED  NO DEFAULT  NOT NULL,
    unit_price   NUMERIC (8,2)         NO DEFAULT  NOT NULL,
    qty_ordered  NUMERIC (5) UNSIGNED  NO DEFAULT  NOT NULL,
    PRIMARY KEY (ordernum, partnum)  );
```

- This example creates a table like the JOB table with the same constraints:

```
CREATE TABLE PERSNL.JOB_CORPORATE
  LIKE PERSNL.JOB WITH CONSTRAINTS;
```

- This is an example of NOT CASESPECIFIC usage:

```
CREATE TABLE T (a char(10) NOT CASESPECIFIC, b char(10));
INSERT INTO T values ('a', 'A');
```

- A row is not returned in this example. Constant 'A' is case sensitive, whereas column 'a' is insensitive.

```
SELECT * FROM T WHERE a = 'A';
```

- The row is returned in this example. Both sides are case sensitive.

```
SELECT * FROM T WHERE a = 'A' (not casespecific);
```

- The row is returned in this example. A case sensitive comparison is done because column 'b' is case sensitive.

```
SELECT * FROM T WHERE b = 'A';
```

- The row is returned in this example. A case sensitive comparison is done because column 'b' is case sensitive.

```
SELECT * FROM T WHERE b = 'A' (not casespecific);
```

Examples of CREATE TABLE AS

This section shows the column attribute rules used to generate and specify the column names and data types of the table being created.

- If *column-attributes* are not specified, the select list items of the select-query are used to generate the column names and data attributes of the created table. If the select list item is a column, then it is used as the name of the created column. For example:

  ```
  create table t as select a,b from t1
  ```

  Table t has 2 columns named (a,b) and the same data attributes as columns from table t1.

- If the select list item is an expression, it must be renamed with an AS clause. An error is returned if expressions are not named. For example:

  ```
  create table t as select a+1 as c from t1
  ```

  Table t has 1 column named (c) and data attribute of (a+1)

  ```
  create table t as select a+1 from t1
  ```

  An error is returned, expression must be renamed.

- If *column-attributes* are specified and contains *datatype-info*, then they override the attributes of the select items in the select query. These data attributes must be compatible with the corresponding data attributes of the select list items in the select-query.

  ```
  create table t(a int) as select b from t1
  ```

  Table t has one column named "a" with data type "int".

  ```
  create table t(a char(10)) as select a+1 b from t1;
  ```

An error is returned because the data attribute of column “a”, a char, does not match the data attribute of the select list item “b” a numeric.

- If *column-attributes* are specified and they only contain *column-name*, then the specified column-name override any name that was derived from the select query.

  ```
  create table t(c,d) as select a,b from t1
  ```

  Table t has 2 columns, c and d, which has the data attributes of columns a and b from table t1.

- If *column-attributes* are specified, then they must contain attributes corresponding to all select list items in the *select-query*. An error is returned, if a mismatch exists.

  ```
  create table t(a int) as select b,c from t1
  ```

  An error is returned. Two items need to be specified as part of the table-attributes.

- The *column-attributes* must specify either the *column-name datatype-info* pair or just the *column-name* for all columns. You cannot specify some columns with just the name and others with name and data type.

  ```
  create table t(a int, b) as select c,d from t1
  ```

  An error is returned.

In the following example, table `t1` is created. Table `t2` is created using the CREATE TABLE AS syntax without table attributes:

```
CREATE TABLE t1 (c1 int not null primary key,
                 c2 char(50));

CREATE TABLE t2 (c1 int, c2 char (50) UPSHIFT NOT NULL)
                AS SELECT * FROM t1;
```

CREATE VIEW Statement

- "Syntax Description of CREATE VIEW"
- "Considerations for CREATE VIEW"
- "Examples of CREATE VIEW"

The CREATE VIEW statement creates a Trafodion SQL view. See "Views" (page 255).

NOTE: DDL statements are not currently supported in transactions. That means that you cannot run this statement inside a user-defined transaction (BEGIN WORK...COMMIT WORK) or when AUTOCOMMIT is OFF. To run this statement, AUTOCOMMIT must be turned ON (the default) for the session.

```
CREATE [OR REPLACE] VIEW view
   [(column-name ] [,column-name ...)]
      AS query-expr [order-by-clause]
   [WITH CHECK OPTION]
```

Syntax Description of CREATE VIEW

OR REPLACE

creates a view if one does not exist or replaces a view if a view of the same name exists. The view being replaced might have the same view definition or a different view definition.

view

specifies the ANSI logical name of the view. See "Database Object Names" (page 198). This name must be unique among names of tables and views within its schema.

(*column-name* [,*column-name*]...)

specifies names for the columns of the view. Column names in the list must match one-for-one with columns in the table specified by *query-expr*.

If you omit this clause, columns in the view have the same names as the corresponding columns in *query-expr*. You must specify this clause if any two columns in the table specified by *query-expr* have the same name or if any column of that table does not have a name. For example, this query expression `SELECT MAX(salary), AVG(salary) AS average_salary FROM employee` the first column does not have a name.

No two columns of the view can have the same name; if a view refers to more than one table and the select list refers to columns from different tables with the same name, you must specify new names for columns that would otherwise have duplicate names.

AS *query-expr*

specifies the columns for the view and sets the selection criteria that determines the rows that make up the view. For information about character string literals, see "Character String Literals" (page 224). For the syntax and syntax description of *query-expr*, see "SELECT Statement" (page 138). The CREATE VIEW statement provides this restriction with regard to the *query-expr* syntax: `[ANY N]`, `[FIRST N]` select list items are not allowed in a view.

order-by-clause

specifies the order in which to sort the rows of the final result table. For the syntax and syntax description of the *order-by-clause*, see "SELECT Statement" (page 138). The CREATE VIEW statement restricts the *order-by-clause* with regard to the *access-clause* and *mode-clause*. The *access-mode* and *mode-clause* cannot follow the *order-by-clause*.

WITH CHECK OPTION

specifies that no row can be inserted or updated in the database through the view unless the row satisfies the view definition—that is, the search condition in the WHERE clause of the query

expression must evaluate to true for any row that is inserted or updated. This option is only allowed for updatable views.

If you omit this option, a newly inserted row or an updated row need not satisfy the view definition, which means that such a row can be inserted or updated in the table but does not appear in the view. This check is performed each time a row is inserted or updated.

WITH CHECK OPTION does not affect the query expression; rows must always satisfy the view definition.

Considerations for CREATE VIEW

- If you specify CREATE OR REPLACE VIEW:
  - A new view is created if a view of the same name does not exist.
  - If a view of same name exists, the old view definition is dropped, and a view with a new definition is created. No check will be done to see if the new view is identical to the view it is replacing. The CREATE OR REPLACE VIEW command will unilaterally drop the old view definition and replace it with the new view definition.
  - The privileges granted on the old view will be re-granted on the new view. If the re-grant of privileges fails, the CREATE OR REPLACE VIEW operation fails.
  - When CREATE OR REPLACE VIEW replaces an existing view, any dependent views will be dropped.
- You can specify GROUP BY using ordinals to refer to the relative position within the SELECT list. For example, `GROUP BY 3, 2, 1.`
- Dynamic parameters are not allowed.

Effect of Adding a Column on View Definitions

The addition of a column to a table has no effect on any existing view definitions or conditions included in constraint definitions. Any implicit column references specified by SELECT * in view or constraint definitions are replaced by explicit column references when the definition clauses are originally evaluated.

Authorization and Availability Requirements

To issue a CREATE VIEW statement, you must have SELECT privileges on the objects underlying the view or be the owner of the objects underlying the view, and one of the following must be true:

- You are DB__ROOT.
- You are creating the view in a shared schema.
- You are the private schema owner.
- You have the CREATE or CREATE_VIEW component privilege for the SQL_OPERATIONS component.

NOTE: In this case, if you create a view in a private schema, it will be owned by the schema owner.

When you create a view on a single table, the owner of the view is automatically given all privileges WITH GRANT OPTION on the view. However, when you create a view that spans multiple tables, the owner of the view is given only SELECT privileges WITH GRANT OPTION. If you try to grant privileges to another user on the view other than SELECT, you will receive a warning that you lack the grant option for that privilege.

Updatable and Non-Updatable Views

Single table views can be updatable. Multi-table views cannot be updatable.

To define an updatable view, a query expression must also meet these requirements:

- It cannot contain a JOIN, UNION, or EXCEPT clause.
- It cannot contain a GROUP BY or HAVING clause.
- It cannot directly contain the keyword DISTINCT.
- The FROM clause must refer to exactly one table or one updatable view.
- It cannot contain a WHERE clause that contains a subquery.
- The select list cannot include expressions or functions or duplicate column names.

ORDER BY Clause Guidelines

The ORDER BY clause can be specified in the SELECT portion of a CREATE VIEW definition. Any SELECT syntax that is valid when the SELECT portion is specified on its own is also valid during the view definition. An ORDER BY clause can contain either the column name from the SELECT list or from *select-list-index*.

When a DML statement is issued against the view, the rules documented in the following sections are used to apply the ORDER BY clause.

When to Use ORDER BY

An ORDER BY clause is used in a view definition only when the clause is under the root of the Select query that uses that view. If the ORDER BY clause appears in other intermediate locations or in a subquery, it is ignored.

Consider this CREATE VIEW statement:

```
create view v as select a from t order by a;
select * from v x, v y;
```

Or this INSERT statement:

```
insert into t1 select * from v;
```

In these two examples, the ORDER BY clause is ignored during DML processing because the first appears as part of a derived table and the second as a subquery selects, both created after the view expansion.

If the same query is issued using explicit derived tables instead of a view, a syntax error is returned:

```
select * from (select a from t order by a) x, (select a from t order by a) y;
```

This example returns a syntax error because an ORDER BY clause is not supported in a subquery.

The ORDER BY clause is ignored if it is part of a view and used in places where it is not supported. This is different than returning an error when the same query was written with explicit ORDER BY clause, as is shown in the preceding examples.

ORDER BY in a View Definition With No Override

If the SELECT query reads from the view with no explicit ORDER BY override, the ORDER BY semantics of the view definition are used.

In this example, the ordering column is the one specified in the CREATE VIEW statement:

```
create view v as select * from t order by a
Select * from v
```

The SELECT query becomes equivalent to:

```
select * from t order by a;
```

ORDER BY in a View Definition With User Override

If a SELECT query contains an explicit ORDER BY clause, it overrides the ORDER BY clause specified in the view definition.

For example:

```
create view v as select a,b from t order by a;
select * from v order by b;
```

In this example, `order by b` overrides the `order by a` specified in the view definition.

The SELECT query becomes equivalent to:

```
select a,b from t order by b;
```

Nested View Definitions

In case of nested view definitions, the ORDER BY clause in the topmost view definition overrides the ORDER BY clause of any nested view definitions.

For example:

```
create view v1 as select a,b from t1 order by a;
create view v2 as select  a,b from v1 order by b;
select * from v2;
```

In this example, the ORDER BY specified in the definition of view `v2` overrides the ORDER BY specified in the definition of view `v1`.

The SELECT query becomes equivalent to:

```
select a,b from (select a, b from t) x order by b;
```

Examples of CREATE VIEW

- This example creates a view on a single table without a view column list:

```
CREATE VIEW SALES.MYVIEW1 AS
  SELECT ordernum, qty_ordered FROM SALES.ODETAIL;
```

- This example replaces the view, MYVIEW1, with a different view definition:

```
CREATE OR REPLACE VIEW SALES.MYVIEW1 AS
  SELECT ordernum, qty_ordered FROM SALES.ODETAIL
  WHERE unit_price > 100;
```

- This example creates a view with a column list:

```
CREATE VIEW SALES.MYVIEW2
  (v_ordernum, t_partnum) AS
  SELECT v.ordernum, t.partnum
    FROM SALES.MYVIEW1 v, SALES.ODETAIL t;
```

- This example creates a view from two tables by using an INNER JOIN:

```
CREATE VIEW MYVIEW4
  (v_ordernum, v_partnum) AS
  SELECT od.ordernum, p.partnum
    FROM SALES.ODETAIL OD INNER JOIN SALES.PARTS P
    ON od.partnum = p.partnum;
```

Vertical Partition Example

This example creates three logical vertical partitions for a table, `vp0`, `vp1`, and `vp2` and then creates a view `vp` to access them.

A view can be used to obtain a composite representation of a set of closely related tables. In the following example tables `vp0`, `vp1` and `vp2` all have a key column `a`. This key column is known to contain identical rows for all three tables. The three tables `vp0`, `vp1` and `vp2` also contain columns `b`, `c` and `d` respectively. We can create a view `vp` that combines these three tables and provides the interface of columns `a`, `b`, `c` and `d` belonging to a single object.

Trafodion SQL has the ability to eliminate redundant joins in a query. Redundant joins occur when:

- Output of join contains expressions from only one of its two children
- Every row from this child will match one and only one row from the other child

Suppose tables A and B denote generic tables. To check if the rule "every row from this child will match one and only one row from the other child" is true, Trafodion SQL uses the fact that the join of Table A with table or subquery B preserves all the rows of A if the join predicate contains an equi-join predicate that references a key of B, and one of the following is true: The join is a left outer join where B is the inner table. In this example, for the join between `vp0` and `vp1`, `vp0` fills the role of table A and `vp1` fills the role of table B. For the join between `vp1` and `vp2`, `vp1` fills the role of table A and `vp2` fills the role of table B.

The view `vp` shown in this example uses left outer joins to combine the three underlying tables. Therefore, if the select list in a query that accesses `vp` does not contain column `d` from `vp2` then the join to table `vp2` in the view `vp` will not be performed.

```
create table vp0(a integer not null, b integer, primary key(a));
create table vp1(a integer not null, c integer, primary key(a));
create table vp2(a integer not null, d integer, primary key(a));

create view vp(a,b,c,d) as
select vp0.a, b, c, d
from vp0 left outer join vp1 on vp0.a=vp1.a
         left outer join vp2 on vp0.a=vp2.a;

select a, b from vp;  -- reads only vp0
select a, c from vp;  -- reads vp0 and vp1
select d from vp; -- reads vp0 and vp2
```

DELETE Statement

- "Syntax Description of DELETE"
- "Considerations for DELETE"
- "Examples of DELETE"

The DELETE statement is a DML statement that deletes a row or rows from a table or an updatable view. Deleting rows from a view deletes the rows from the table on which the view is based. DELETE does not remove a table or view, even if you delete the last row in the table or view.

Trafodion SQL provides searched DELETE—deletes rows whose selection depends on a search condition.

For the searched DELETE form, if no WHERE clause exists, all rows are deleted from the table or view.

```
Searched DELETE is:

DELETE FROM table

[WHERE search-condition ]

   [[FOR] access-option ACCESS]

access-option is:
   READ COMMITTED
```

Syntax Description of DELETE

table

names the user table or view from which to delete rows. *table* must be a base table or an updatable view. To refer to a table or view, use the ANSI logical name.

See "Database Object Names" (page 198).

WHERE *search-condition*

specifies a search condition that selects rows to delete. Within the search condition, any columns being compared are columns in the table or view being deleted from. See "Search Condition" (page 250).

If you do not specify a search condition, all rows in the table or view are deleted.

[FOR] *access-option* ACCESS

specifies the access option required for data used to evaluate the search condition. See "Data Consistency and Access Options" (page 25).

READ COMMITTED

specifies that any data used to evaluate the search condition must come from committed rows.

The default access option is the isolation level of the containing transaction.

Considerations for DELETE

Authorization Requirements

DELETE requires authority to read and write to the table or view being deleted from and authority to read tables or views specified in subqueries used in the search condition.

Transaction Initiation and Termination

The DELETE statement automatically initiates a transaction if no transaction is active. Otherwise, you can explicitly initiate a transaction with the BEGIN WORK statement. When a transaction is

started, the SQL statements execute within that transaction until a COMMIT or ROLLBACK is encountered or an error occurs.

Isolation Levels of Transactions and Access Options of Statements

The isolation level of an SQL transaction defines the degree to which the operations on data within that transaction are affected by operations of concurrent transactions. When you specify access options for the DML statements within a transaction, you override the isolation level of the containing transaction. Each statement then executes with its individual access option.

Examples of DELETE

- Remove all rows from the JOB table:

```
DELETE FROM persnl.job;

--- 10 row(s) deleted.
```

- Remove from the table ORDERS any orders placed with sales representative 220 by any customer except customer number 1234:

```
DELETE FROM sales.orders
WHERE salesrep = 220 AND custnum <> 1234;

--- 2 row(s) deleted.
```

- Remove all suppliers not in Texas from the table PARTSUPP:

```
DELETE FROM invent.partsupp
WHERE suppnum IN
  (SELECT suppnum FROM samdbcat.invent.supplier
   WHERE state <> 'TEXAS');

--- 41 row(s) deleted.
```

  This statement achieves the same result:

```
DELETE FROM invent.partsupp
WHERE suppnum NOT IN
  (SELECT suppnum FROM samdbcat.invent.supplier
   WHERE state = 'TEXAS');

--- 41 row(s) deleted.
```

- This is an example of a self-referencing DELETE statement, where the table from which rows are deleted is scanned in a subquery:

```
delete from table1 where a in
(select a from table1 where b > 200)
```

DROP FUNCTION Statement

- "Syntax Description of DROP FUNCTION"
- "Considerations for DROP FUNCTION"
- "Examples of DROP FUNCTION"

The DROP FUNCTION statement removes a user-defined function (UDF) from the Trafodion database.

NOTE: DDL statements are not currently supported in transactions. That means that you cannot run this statement inside a user-defined transaction (BEGIN WORK...COMMIT WORK) or when AUTOCOMMIT is OFF. To run this statement, AUTOCOMMIT must be turned ON (the default) for the session.

```
DROP FUNCTION [[catalog-name.]schema-name.]function-name
```

Syntax Description of DROP FUNCTION

[[*catalog-name*.]*schema-name*.]*function-name*

specifies the ANSI logical name of the function, where each part of the name is a valid SQL identifier with a maximum of 128 characters. Specify the name of a function that has already been registered in the schema. If you do not fully qualify the function name, Trafodion SQL qualifies it according to the schema of the current session. For more information, see "Identifiers" (page 221) and "Database Object Names" (page 198).

Considerations for DROP FUNCTION

Required Privileges

To issue a DROP FUNCTION statement, one of the following must be true:

- You are DB__ROOT.
- You are the owner of the function.
- You have the DROP or DROP_ROUTINE component privilege for SQL_OPERATIONS component.

Examples of DROP FUNCTION

- This DROP FUNCTION statement removes the function named ADD2 from the default schema:

  ```
  DROP FUNCTION add2;
  ```

- This DROP FUNCTION statement removes the function named MMA5 from the default schema:

  ```
  DROP PROCEDURE mma5;
  ```

- This DROP FUNCTION statement removes the function named REVERSE from the default schema:

  ```
  DROP PROCEDURE reverse;
  ```

DROP INDEX Statement

- "Syntax Description of DROP INDEX"
- "Considerations for DROP INDEX"
- "Examples of DROP INDEX"

The DROP INDEX statement drops a Trafodion SQL index. See "Indexes" (page 222).

DROP INDEX is a Trafodion SQL extension.

NOTE: DDL statements are not currently supported in transactions. That means that you cannot run this statement inside a user-defined transaction (BEGIN WORK...COMMIT WORK) or when AUTOCOMMIT is OFF. To run this statement, AUTOCOMMIT must be turned ON (the default) for the session.

```
DROP [VOLATILE] INDEX index
```

Syntax Description of DROP INDEX

index

is the index to drop.

For information, see "Database Object Names" (page 198).

Considerations for DROP INDEX

Required Privileges

To issue a DROP INDEX statement, one of the following must be true:

- You are DB__ROOT.
- You are the owner of the index or the table associated with the index.
- You have the DROP or DROP_INDEX component privilege for the SQL_OPERATIONS component.

Examples of DROP INDEX

- This example drops an index:

```
DROP INDEX myindex;
```

- This example drops a volatile index:

```
DROP VOLATILE INDEX vindex;
```

DROP LIBRARY Statement

- "Syntax Description of DROP LIBRARY"
- "Considerations for DROP LIBRARY"
- "Examples of DROP LIBRARY"

The DROP LIBRARY statement removes a library object from the Trafodion database and also removes the library file referenced by the library object.

DROP LIBRARY is a Trafodion SQL extension.

NOTE: DDL statements are not currently supported in transactions. That means that you cannot run this statement inside a user-defined transaction (BEGIN WORK...COMMIT WORK) or when AUTOCOMMIT is OFF. To run this statement, AUTOCOMMIT must be turned ON (the default) for the session.

```
DROP LIBRARY [[catalog-name.]schema-name.]library-name [RESTRICT | CASCADE]
```

Syntax Description of DROP LIBRARY

[[*catalog-name*.]*schema-name*.]*library-name*

specifies the ANSI logical name of the library object, where each part of the name is a valid SQL identifier with a maximum of 128 characters. Specify the name of a library object that has already been registered in the schema. If you do not fully qualify the library name, Trafodion SQL qualifies it according to the schema of the current session. For more information, see "Identifiers" (page 221) and "Database Object Names" (page 198).

[RESTRICT | CASCADE]

If you specify RESTRICT, the DROP LIBRARY operation fails if any stored procedures in Java (SPJs) or user-defined functions (UDFs) were created based on the specified library.

If you specify CASCADE, any such dependent procedures or functions are removed as part of the DROP LIBRARY operation.

The default value is RESTRICT.

Considerations for DROP LIBRARY

- RESTRICT requires that all procedures and functions that refer to the library object be dropped before you drop the library object. CASCADE automatically drops any procedures or functions that are using the library.
- If the library filename referenced by the library object does not exist, Trafodion SQL issues a warning.

Required Privileges

To issue a DROP LIBRARY statement, one of the following must be true:

- You are DB__ROOT.
- You are the owner of the library.
- You have the DROP or DROP_LIBRARY component privilege for the SQL_OPERATIONS component.

Examples of DROP LIBRARY

- This DROP LIBRARY statement removes the library named SALESLIB from the SALES schema, removes the `Sales2.jar` file referenced by the library, and drops any stored procedures in Java (SPJs) that were created based on this library:

```
DROP LIBRARY sales.saleslib CASCADE;
```

- This DROP LIBRARY statement removes the library named MYUDFS from the default schema and removes the $TMUDFS library file referenced by the library:

  ```
  DROP LIBRARY myudfs RESTRICT;
  ```

  RESTRICT prevents the DROP LIBRARY operation from dropping any user-defined functions (UDFs) that were created based on this library. If any UDFs were created based on this library, the DROP LIBRARY operation fails.

DROP PROCEDURE Statement

- "Syntax Description of DROP PROCEDURE"
- "Considerations for DROP PROCEDURE"
- "Examples of DROP PROCEDURE"

The DROP PROCEDURE statement removes a stored procedure in Java (SPJ) from the Trafodion database.

NOTE: DDL statements are not currently supported in transactions. That means that you cannot run this statement inside a user-defined transaction (BEGIN WORK...COMMIT WORK) or when AUTOCOMMIT is OFF. To run this statement, AUTOCOMMIT must be turned ON (the default) for the session.

```
DROP PROCEDURE [[catalog-name.]schema-name.]procedure-name
```

Syntax Description of DROP PROCEDURE

[[*catalog-name*.]*schema-name*.]*procedure-name*

specifies the ANSI logical name of the stored procedure in Java (SPJ), where each part of the name is a valid SQL identifier with a maximum of 128 characters. Specify the name of a procedure that has already been registered in the schema. If you do not fully qualify the procedure name, Trafodion SQL qualifies it according to the schema of the current session. For more information, see "Identifiers" (page 221) and "Database Object Names" (page 198).

Considerations for DROP PROCEDURE

Required Privileges

To issue a DROP PROCEDURE statement, one of the following must be true:

- You are DB__ROOT.
- You are the owner of the procedure.
- You have the DROP or DROP_ROUTINE component privilege for SQL_OPERATIONS component.

Examples of DROP PROCEDURE

- This DROP PROCEDURE statement removes the procedure named LOWERPRICE from the SALES schema:

  ```
  DROP PROCEDURE sales.lowerprice;
  ```

- This DROP PROCEDURE statement removes the procedure TOTALPRICE from the default schema for the session, which is the SALES schema:

  ```
  SET SCHEMA sales;
  DROP PROCEDURE totalprice;
  ```

DROP ROLE Statement

- "Syntax Description of DROP ROLE"
- "Considerations for DROP ROLE"
- "Examples of DROP ROLE"

The DROP ROLE statement deletes an SQL role. See "Roles" (page 248).

```
DROP ROLE role-name
```

Syntax Description of DROP ROLE

role-name

is an existing role name. The role cannot be dropped if any of the following are true:

- Any privileges are granted to the role.
- The role is granted to any users.
- The role owns any schemas.

Considerations for DROP ROLE

- To drop a role, you must own the role or have user administrative privileges for the role. You have user administrative privileges for the role if you have been granted the MANAGE_ROLES component privilege. Initially, DB__ROOT is the only database user who has been granted the MANAGE_ROLES component privilege.
- Role names beginning with DB__ are reserved and can only be dropped by DB__ROOT.
- You can determine all users to whom a role has been granted by using the SHOWDDL ROLE statement. See the "SHOWDDL Statement" (page 160).

Before You Drop a Role

Before dropping a role, follow these guidelines:

- You must revoke all privileges granted to the role.
- You must revoke the role from all users to whom it was granted.
- You must drop all schemas the role is a manager (or owner) of.

You can determine all users to whom a role has been granted with the SHOWDDL statement. See the "SHOWDDL Statement" (page 160).

Active Sessions for the User

In Trafodion Release 0.9, when you revoke a role from a user, the effects on any active sessions for the user are undefined. We recommend that you disconnect such sessions. The user then reconnects to establish new sessions with the updated set of privileges.

Starting in Trafodion Release 1.0, when you revoke a role from a user, the change in privileges is automatically propagated to and detected by active sessions. There is no need for users to disconnect from and reconnect to a session to see the updated set of privileges.

Examples of DROP ROLE

- To drop a role:

  ```
  DROP ROLE clerks;
  ```

- To drop a role with dependent privileges:

```
-- User administrator creates a role:
CREATE ROLE clerks;
-- User administrator grants privileges on a table to the role:
GRANT ALL PRIVILEGES ON TABLE invent.partloc TO clerks;
-- User administrator grants the role to a user:
GRANT ROLE clerks TO JSmith;
-- JSmith creates a view based upon the granted privilege:
CREATE VIEW invent.partlocView (partnum, loc_code)
  AS SELECT partnum, loc_code FROM invent.partloc;
-- If the user administrator attempts to drop the role, this
--   would fail because of the view created based on
--   the granted privilege.
-- To successfully drop the role, the dependent view
--   and grant must be removed first. For this example:
-- 1. JSmith drops the view:
DROP VIEW invent.partlocView;
-- 2. User administrator revokes the role from the user:
REVOKE ROLE clerks FROM JSmith;
-- 3. User administrator revokes all privileges the role has been granted
REVOKE ALL ON invent.partloc FROM clerks;
-- 4. User administrator drops the role:
DROP ROLE clerks;
-- The DROP ROLE operation succeeds.
```

DROP SCHEMA Statement

- "Syntax Description of DROP SCHEMA"
- "Considerations for DROP SCHEMA"
- "Example of DROP SCHEMA"

The DROP SCHEMA statement drops a schema from the database. See"Schemas" (page 249).

NOTE: DDL statements are not currently supported in transactions. That means that you cannot run DDL statements inside a user-defined transaction (BEGIN WORK...COMMIT WORK) or when AUTOCOMMIT is OFF. To run these statements, AUTOCOMMIT must be turned ON (the default) for the session.

```
DROP SCHEMA schema-name [RESTRICT|CASCADE]
```

Syntax Description of DROP SCHEMA

schema-name
: is the name of the schema to delete.

RESTRICT
: If you specify RESTRICT, an error is reported if the specified schema is not empty. The default is RESTRICT.

CASCADE
: If you specify CASCADE, objects in the specified schema and the schema itself are dropped. Any objects in other schemas that were dependent on objects in this schema are dropped as well.

Considerations for DROP SCHEMA

Authorization Requirements

To drop a schema, one of the following must be true:

- You are the owner of the schema.
- You have been granted the role that owns the schema.
- You have been granted the DROP_SCHEMA privilege.

Example of DROP SCHEMA

This example drops an empty schema:

```
DROP SCHEMA sales;
```

DROP TABLE Statement

- "Syntax Description of DROP TABLE"
- "Considerations for DROP TABLE"
- "Examples of DROP TABLE"

The DROP TABLE statement deletes a Trafodion SQL table and its dependent objects such as indexes and constraints. See "Tables" (page 254).

NOTE: DDL statements are not currently supported in transactions. That means that you cannot run this statement inside a user-defined transaction (BEGIN WORK...COMMIT WORK) or when AUTOCOMMIT is OFF. To run this statement, AUTOCOMMIT must be turned ON (the default) for the session.

```
DROP [VOLATILE] TABLE [IF EXISTS] table [RESTRICT|CASCADE]
```

Syntax Description of DROP TABLE

VOLATILE
: specifies that the table to be dropped is a volatile table.

IF EXISTS
: drops the HBase table if it exists. This option does not apply to volatile tables.

table
: is the name of the table to delete.

RESTRICT
: If you specify RESTRICT and the table is referenced by another object, the specified table cannot be dropped. The default is RESTRICT.

CASCADE
: If you specify CASCADE, the table and all objects referencing the table (such as a view) are dropped.

Considerations for DROP TABLE

Authorization Requirements

To issue a DROP TABLE statement, one of the following must be true:

- You are DB__ROOT.
- You are the owner of the table.
- You have the DROP or DROP_TABLE component privilege for the SQL_OPERATIONS component.

Examples of DROP TABLE

- This example drops a table:

```
DROP TABLE mysch.mytable;
```

- This example drops a volatile table:

```
DROP VOLATILE TABLE vtable;
```

DROP VIEW Statement

- "Syntax Description of DROP VIEW"
- "Considerations for DROP VIEW"
- "Example of DROP VIEW"

The DROP VIEW statement deletes a Trafodion SQL view. See "Views" (page 255).

NOTE: DDL statements are not currently supported in transactions. That means that you cannot run this statement inside a user-defined transaction (BEGIN WORK...COMMIT WORK) or when AUTOCOMMIT is OFF. To run this statement, AUTOCOMMIT must be turned ON (the default) for the session.

```
DROP VIEW view [RESTRICT|CASCADE]
```

Syntax Description of DROP VIEW

view
is the name of the view to delete.

RESTRICT
If you specify RESTRICT, you cannot drop the specified view if it is referenced in the query expression of any other view or in the search condition of another object's constraint. The default is RESTRICT.

CASCADE
If you specify CASCADE, any dependent objects are dropped.

Considerations for DROP VIEW

Authorization Requirements

To issue a DROP VIEW statement, one of the following must be true:

- You are DB__ROOT.
- You are the owner of the view.
- You have the DROP or DROP_VIEW component privilege for the SQL_OPERATIONS component.

Example of DROP VIEW

This example drops a view:

```
DROP VIEW mysch.myview;
```

EXECUTE Statement

- "Syntax Description of EXECUTE"
- "Considerations for EXECUTE"
- "Examples of EXECUTE"

The EXECUTE statement executes an SQL statement previously compiled by a PREPARE statement in a Trafodion Command Interface (TrafCI) session.

```
EXECUTE statement-name
   [ USING param [,param]...] ]

param is:
   ?param-name | literal-value
```

Syntax Description of EXECUTE

statement-name

is the name of a prepared SQL statement—that is, the statement name used in the PREPARE statement. *statement-name* is an SQL identifier. See "Identifiers" (page 221).

USING *param* [,*param*]... *param* is: ?*param-name* | *literal-value*

specifies values for unnamed parameters (represented by ?) in the prepared statement in the form of either a parameter name (?*param-name*) or a literal value (*literal-value*). The data type of a parameter value must be compatible with the data type of the associated parameter in the prepared statement.

Parameter values (*param*) are substituted for unnamed parameters in the prepared statement by position—the i-th value in the USING clause is the value for the i-th parameter in the statement. If fewer parameter values exist in the USING clause than unnamed parameters in the PREPARE statement, Trafodion SQL returns an error. If more parameter values exist in the USING clause than the unnamed parameters in the PREPARE statement, Trafodion SQL issues warning 15019.

The USING clause does not set parameter values for named parameters (represented by ?*param-name*) in a prepared statement. To set parameter values for named parameters, use the SET PARAM command. For more information, see the *Trafodion Command Interface Guide*.

?*param-name*

The value for a ?*param-name* must be previously specified with the SET PARAM command. The *param-name* is case-sensitive. For information about the SET PARAM command, see the *Trafodion Command Interface Guide*.

literal-value

is a numeric or character literal that specifies the value for the unnamed parameter.

If *literal-value* is a character literal and the target column type is character, you do not have to enclose it in single quotation marks. Its data type is determined from the data type of the column to which the literal is assigned. If the *literal-value* contains leading or trailing spaces, commas, or if it matches any parameter names that are already set, enclose the *literal-value* in single quotes.

See the "PREPARE Statement" (page 126). For information about the SET PARAM command, see the *Trafodion Command Interface Guide*.

Considerations for EXECUTE

Scope of EXECUTE

A statement must be compiled by PREPARE before you EXECUTE it, but after it is compiled, you can execute the statement multiple times without recompiling it. The statement must have been compiled during the same TrafCI session as its execution.

Examples of EXECUTE

- Use PREPARE to compile a statement once, and then execute the statement multiple times with different parameter values. This example uses the SET PARAM command to set parameter values for named parameters (represented by `?param-name`) in the prepared statement.

```
SQL>prepare findemp from
+>select * from persnl.employee
+>where salary > ?sal and jobcode = ?job;

--- SQL command prepared.

SQL>set param ?sal 40000.00;

SQL>set param ?job 450;

SQL>execute findemp;

EMPNUM FIRST_NAME      LAST_NAME      DEPTNUM JOBCODE SALARY
------ --------------- -------------- ------- ------- --------
   232 THOMAS          SPINNER           4000     450 45000.00

--- 1 row(s) selected.

SQL>set param ?sal 20000.00;

SQL>set param ?job 300;

SQL>execute findemp;

EMPNUM FIRST_NAME      LAST_NAME      DEPTNUM JOBCODE SALARY
------ --------------- -------------- ------- ------- --------
    75 TIM             WALKER            3000     300 32000.00
    89 PETER           SMITH             3300     300 37000.40
...

--- 13 row(s) selected.
```

- Specify literal values in the USING clause of the EXECUTE statement for unnamed parameters in the prepared statement:

```
SQL>prepare findemp from
+>select * from persnl.employee
+>where salary > ? and jobcode = ?;

--- SQL command prepared.

SQL>execute findemp using 40000.00,450;

EMPNUM FIRST_NAME      LAST_NAME      DEPTNUM JOBCODE SALARY
------ --------------- -------------- ------- ------- --------
   232 THOMAS          SPINNER           4000     450 45000.00

--- 1 row(s) selected.
```

```
SQL>execute findemp using 20000.00, 300;

EMPNUM FIRST_NAME      LAST_NAME      DEPTNUM JOBCODE SALARY
------ --------------- -------------- ------- ------- --------
    75 TIM             WALKER            3000     300 32000.00
    89 PETER           SMITH             3300     300 37000.40
...

--- 13 row(s) selected.
```

- Use SET PARAM to assign a value to a parameter name and specify both the parameter name and a literal value in the EXECUTE USING clause:

```
SQL>prepare findemp from
+>select * from persnl.employee
+>where salary > ? and jobcode = ?;

--- SQL command prepared.

SQL>set param ?Salary 40000.00;

SQL>execute findemp using ?Salary, 450;

EMPNUM FIRST_NAME      LAST_NAME      DEPTNUM JOBCODE SALARY
------ --------------- -------------- ------- ------- --------
232    THOMAS          SPINNER           4000     450 45000.00
```

EXPLAIN Statement

The EXPLAIN statement helps you to review query execution plans. You can use the EXPLAIN statement anywhere you can execute other SQL statements (for example, SELECT). For more information on the EXPLAIN function, see "EXPLAIN Function" (page 342).

EXPLAIN is a Trafodion SQL extension.

```
EXPLAIN [OPTIONS {'f'}] {FOR QID query-text | prepared-stmt-name}
```

Table 2 EXPLAIN Statement Options

Syntax	Option Type	Purpose
`OPTIONS 'f'`	Formatted	Provides the simple, basic information contained in the query execution plan. This information is formatted for readability and limited to 79 characters (one line) per operator.

Plans displayed by the EXPLAIN statement are ordered from top (root operator) to bottom (leaf operators).

Syntax Description of EXPLAIN

f

formatted. See "Formatted [OPTIONS 'f'] Considerations" (page 102).

query-text

a DML statement such as SELECT * FROM T3.

prepared-stmt-name

an SQL identifier containing the name of a statement already prepared in this session. An SQL identifier is case-insensitive (will be in uppercase) unless it is double-quoted. It must be double-quoted if it contains blanks, lowercase letters, or special characters. It must start with a letter. When you refer to the prepared query in a SELECT statement, you must use uppercase.

Considerations for EXPLAIN

- "Required Privileges" (page 101)
- "Obtaining EXPLAIN Plans While Queries Are Running" (page 101)
- "Case Considerations" (page 102)
- "Number Considerations" (page 102)
- "Formatted [OPTIONS 'f'] Considerations" (page 102)

Required Privileges

To issue an EXPLAIN statement, one of the following must be true:

- You are DB__ROOT.
- You own (that is, issued) the query specified in the EXPLAIN statement.
- You have the SHOW component privilege for the SQL_OPERATIONS component. The SHOW component privilege is granted to PUBLIC by default.

Obtaining EXPLAIN Plans While Queries Are Running

Trafodion SQL provides the ability to capture an EXPLAIN plan for a query at any time while the query is running with the FOR QID option. By default, this behavior is disabled for a Trafodion database session.

NOTE: Enable this feature before you start preparing and executing queries.

After the feature is enabled, use the FOR QID option in an EXPLAIN statement to get the query execution plan of a running query.

The EXPLAIN function or statement returns the plan that was generated when the query was prepared. EXPLAIN with the FOR QID option retrieves all the information from the original plan of the executing query. The plan is available until the query finishes executing and is removed or deallocated.

Case Considerations

In most cases, words in the commands can be in uppercase or lowercase. The options letter must be single quoted and in lowercase.

Number Considerations

Costs are given in a generic unit of effort. They show relative costs of an operation.

When trailing decimal digits are zero, they are dropped. For example, 6.4200 would display as 6.42 and 5.0 would display as 5, without a decimal point.

Formatted [OPTIONS 'f'] Considerations

The formatted option is the simplest option. It provides essential, brief information about the plan and shows the operators and their order within the query execution plan.

OPTIONS 'f' formats the EXPLAIN output into these fields:

LC	Left child sequence number
RC	Right child sequence number
OP	The sequence number of the operator in the query plan
OPERATOR	The operator type
OPT	Query optimizations that were applied
DESCRIPTION	Additional information about the operator
CARD	Estimated number of rows returned by the plan. CARDINALITY and ROWS_OUT are the same.

This example uses OPTIONS 'f ':

```
>>explain options 'f' select * from region;

LC   RC   OP   OPERATOR              OPT       DESCRIPTION           CARD
---- ---- ---- --------------------  --------  --------------------  ---------

1    .    2    root                                                  1.00E+002
.    .    1    trafodion_scan                 REGION                 1.00E+002

--- SQL operation complete.
```

To use the EXPLAIN statement with a prepared statement, first prepare the query. Then use the EXPLAIN statement:

```
PREPARE q FROM SELECT * FROM REGION;

EXPLAIN options 'f' q;
```

GET Statement

- "Syntax Description of GET"
- "Considerations for GET"
- "Examples of GET"

The GET statement displays the names of database objects, components, component privileges, roles, or users that exist in the Trafodion instance.

GET is a Trafodion SQL extension.

```
GET option

option is:
     COMPONENT PRIVILEGES ON component-name [FOR auth-name]
   | COMPONENTS
   | FUNCTIONS FOR LIBRARY [[catalog-name.]schema-name.]library-name
   | FUNCTIONS [IN SCHEMA [catalog-name.]schema-name]
   | LIBRARIES [IN SCHEMA [catalog-name.]schema-name]
   | PROCEDURES FOR LIBRARY [[catalog-name.]schema-name.]library-name
   | PROCEDURES [IN SCHEMA [catalog-name.]schema-name]
   | ROLES [FOR USER database-username]
   | SCHEMAS [IN CATALOG catalog-name]
   | SCHEMAS FOR [USER | ROLE] authorization-id
   | TABLES [IN SCHEMA [catalog-name.]schema-name]
   | USERS [FOR ROLE role-name]
   | VIEWS [IN SCHEMA [catalog-name.]schema-name]
   | VIEWS ON TABLE [[catalog-name.]schema-name.]table-name
```

Syntax Description of GET

COMPONENT PRIVILEGES ON *component-name*

displays the names of the component privileges available for the specified component.

COMPONENT PRIVILEGES ON *component-name* FOR *auth-name*

displays the component privileges that have been granted to the specified authorization name for the specified component. The *auth-name* is either a registered database username or an existing role name and can be a regular or delimited case-insensitive identifier. See "Case-Insensitive Delimited Identifiers" (page 221).

COMPONENTS

displays a list of all the existing components.

FUNCTIONS

displays the names of all the user-defined functions (UDFs) in the catalog and schema of the current session. By default, the catalog is TRAFODION, and the schema is SEABASE.

FUNCTIONS FOR LIBRARY [[*catalog-name*.]*schema-name*.]*library-name*

displays the UDFs that reference the specified library.

FUNCTIONS IN SCHEMA [*catalog-name*.]*schema-name*

displays the names of all the UDFs in the specified schema.

LIBRARIES

displays the names of all the libraries in the catalog and schema of the current session. By default, the catalog is TRAFODION, and the schema is SEABASE.

LIBRARIES IN SCHEMA [*catalog-name*.]*schema-name*

displays the libraries in the specified schema.

PROCEDURES

displays the names of all the procedures in the catalog and schema of the current session. By default, the catalog is TRAFODION, and the schema is SEABASE.

PROCEDURES FOR LIBRARY [[*catalog-name*.]*schema-name*.]*library-name*

displays the procedures that reference the specified library.

PROCEDURES IN SCHEMA [*catalog-name*.]*schema-name*

displays the names of all the procedures in the specified schema.

ROLES

displays a list of all the existing roles.

ROLES FOR USER *database-username*

displays all the roles that have been granted to the specified database user. The *database-username* can be a regular or delimited case-insensitive identifier. See "Case-Insensitive Delimited Identifiers" (page 221).

SCHEMAS

displays the names of all the schemas in the catalog of the current session. By default, the catalog is TRAFODION.

SCHEMAS IN CATALOG *catalog-name*

displays the names of all the schemas in the specified catalog. For the *catalog-name*, you can specify only TRAFODION.

SCHEMAS FOR [USER | ROLE] *authorization-id*

displays all the schemas managed (or owned) by a specified user or role. *authorization-id* is the name of a user or role. You may specify either USER or ROLE for users or roles.

TABLES

displays the names of all the tables in the catalog and schema of the current session. By default, the catalog is TRAFODION, and the schema is SEABASE.

TABLES IN SCHEMA [*catalog-name*.]*schema-name*

displays the names of all the tables in the specified schema.

USERS

displays a list of all the registered database users.

USERS FOR ROLE *role-name*

displays all the database users who have been granted the specified role. The *role-name* can be a regular or delimited case-insensitive identifier. See "Case-Insensitive Delimited Identifiers" (page 221).

VIEWS

displays the names of all the views in the catalog and schema of the current session. By default, the catalog is TRAFODION, and the schema is SEABASE.

VIEWS IN SCHEMA [*catalog-name*.]*schema-name*

displays the names of all the views in the specified schema. For the *catalog-name*, you can specify only TRAFODION.

VIEWS ON TABLE [[*catalog-name*.]*schema-name*.]*table-name*

displays the names of all the views that were created for the specified table. If you do not qualify the table name with catalog and schema names, GET uses the catalog and schema of the current session. For the *catalog-name*, you can specify only TRAFODION.

Considerations for GET

IMPORTANT: The GET COMPONENT PRIVILEGES, GET COMPONENTS, GET ROLES FOR USER, and GET USERS FOR ROLE statements work only when authentication and authorization are enabled in Trafodion. For more information, see Enabling Security Features in the Trafodion wiki.

The GET statement displays delimited object names in their internal format. For example, the GET statement returns the delimited name `"my ""table"""` as `my "table"`.

Required Privileges

To issue a GET statement, one of the following must be true:

- You are DB__ROOT.
- You have the SHOW component privilege for the SQL_OPERATIONS component. The SHOW component privilege is granted to PUBLIC by default.

Examples of GET

- This GET statement displays the names of all the schemas in the catalog of the current session, which happens to be the TRAFODION catalog:

```
GET SCHEMAS;
```

- This GET statement displays the names of all the schemas in the specified catalog, TRAFODION:

```
GET SCHEMAS IN CATALOG TRAFODION;
```

- This GET statement displays the names of schemas owned by DB__ROOT:

```
GET SCHEMAS FOR USER DB__ROOT;
```

- This GET statement displays the names of all the tables in the catalog and schema of the current session, which happens to be TRAFODION.SEABASE:

```
GET TABLES;
```

- This GET statement displays the names of all the tables in the specified schema, SEABASE2, in the TRAFODION catalog:

```
GET TABLES IN SCHEMA SEABASE2;
```

- This GET statement displays the names of all the views in the catalog and schema of the current session, which happens to be TRAFODION.SEABASE:

```
GET VIEWS;
```

- This GET statement displays the names of all the views in the specified schema, SEABASE2, the TRAFODION catalog:

```
GET VIEWS IN SCHEMA SEABASE2;
```

- This GET statement displays the names of all the views that were created for the specified table, T, in the TRAFODION.SEABASE schema:

```
GET VIEWS ON TABLE T;
```

- This GET statement displays the names of the libraries in the catalog and schema of the current session, which happens to be TRAFODION.SEABASE:

```
GET LIBRARIES;
```

- This GET statement displays the names of the libraries in the TRAFODION._MD_ schema:

```
GET LIBRARIES IN SCHEMA "_MD_";
```

- This GET statement displays the names of procedures registered in the library, TRAFODION._MD_.UDR_LIBRARY:

```
GET PROCEDURES FOR LIBRARY "_MD_".UDR_LIBRARY;
```

- This GET statement displays the names of procedures in the TRAFODION._MD_ schema:

```
GET PROCEDURES IN SCHEMA "_MD_";
```

- This GET statement displays the names of procedures in the catalog and schema of the current session, which happens to be TRAFODION.SEABASE:

```
GET PROCEDURES;
```

- This GET statement displays the names of user-defined functions (UDFs) in the catalog and schema of the current session, which happens to be TRAFODION.SEABASE:

  ```
  GET FUNCTIONS;
  ```

- This GET statement displays the names of UDFs in MYSCHEMA:

  ```
  GET FUNCTIONS IN SCHEMA MYSCHEMA;
  ```

- This GET statement displays the names of UDFs created in the library, TRAFODION.MYSCHEMA.MYUDFS:

  ```
  GET FUNCTIONS FOR LIBRARY MYSCHEMA.MYUDFS;
  ```

- This GET statement displays a list of all the existing components:

  ```
  get components;
  ```

- This GET statement displays the names of the component privileges available for the SQL_OPERATIONS component:

  ```
  get component privileges on sql_operations;
  ```

- This GET statement displays the component privileges that have been granted to the DB__ROOT user for the SQL_OPERATIONS component:

  ```
  get component privileges on sql_operations for db__root;
  ```

- This GET statement displays a list of all the existing roles:

  ```
  get roles;
  ```

- This GET statement displays all the roles that have been granted to the DB__ROOT user:

  ```
  get roles for user db__root;
  ```

- This GET statement displays a list of all the registered database users:

  ```
  get users;
  ```

- This GET statement displays all the database users who have been granted the DB__ROOTROLE role:

  ```
  get users for role db__rootrole;
  ```

GET HBASE OBJECTS Statement

- "Syntax Description of GET HBASE OBJECTS" (page 107)
- "Examples of GET HBASE OBJECTS"

The GET HBASE OBJECTS statement displays a list of HBase objects directly from HBase, not from the Trafodion metadata, and it can be run in any SQL interface, such as the Trafodion Command Interface (TrafCI). This command is equivalent to running a `list` command from an HBase shell, but without having to start and connect to an HBase shell.

GET HBASE OBJECTS is a Trafodion SQL extension.

```
GET [ USER | SYSTEM | EXTERNAL | ALL } HBASE OBJECTS
```

Syntax Description of GET HBASE OBJECTS

USER
: displays a list of the Trafodion user objects.

SYSTEM
: displays a list of the Trafodion system objects, such as metadata, repository, privileges, and Distributed Transaction Manager (DTM) tables.

EXTERNAL
: displays a list of non-Trafodion objects.

ALL
: displays a list of all objects, including user, system, and external objects.

Examples of GET HBASE OBJECTS

- This GET HBASE OBJECTS statement displays the Trafodion user objects in HBase:

```
Trafodion Conversational Interface 1.1.0
(c) Copyright 2014 Hewlett-Packard Development Company, LP.
>>get user hbase objects;

TRAFODION.SCH.SB_HISTOGRAMS
TRAFODION.SCH.SB_HISTOGRAM_INTERVALS
TRAFODION.SCH.T006T1
TRAFODION.SCH.T006T2
TRAFODION.SCH.T006T3
TRAFODION.SCH.T006T4
TRAFODION.SCH.T006T5
TRAFODION.SCH.T006T6
TRAFODION.SCH.T006T7
TRAFODION.SCH.T006T8
TRAFODION.SCH.X1
TRAFODION.SCH.X2
TRAFODION.SCH.X3

--- SQL operation complete.
```

- This GET HBASE OBJECTS statement displays the Trafodion system objects in HBase:

```
>>get system hbase objects;

TRAFODION._DTM_.TLOG0_CONTROL_POINT
…
TRAFODION._DTM_.TLOG1_LOG_f
TRAFODION._MD_.AUTHS
TRAFODION._MD_.COLUMNS
TRAFODION._MD_.DEFAULTS
TRAFODION._MD_.INDEXES
```

```
TRAFODION._MD_.KEYS
TRAFODION._MD_.LIBRARIES
TRAFODION._MD_.LIBRARIES_USAGE
TRAFODION._MD_.OBJECTS
TRAFODION._MD_.OBJECTS_UNIQ_IDX
TRAFODION._MD_.REF_CONSTRAINTS
TRAFODION._MD_.ROUTINES
TRAFODION._MD_.SEQ_GEN
TRAFODION._MD_.TABLES
TRAFODION._MD_.TABLE_CONSTRAINTS
TRAFODION._MD_.TEXT
TRAFODION._MD_.UNIQUE_REF_CONSTR_USAGE
TRAFODION._MD_.VERSIONS
TRAFODION._MD_.VIEWS
TRAFODION._MD_.VIEWS_USAGE
TRAFODION._REPOS_.METRIC_QUERY_AGGR_TABLE
TRAFODION._REPOS_.METRIC_QUERY_TABLE
TRAFODION._REPOS_.METRIC_SESSION_TABLE
TRAFODION._REPOS_.METRIC_TEXT_TABLE

--- SQL operation complete.
```

- This GET HBASE OBJECTS statement displays the external, non-Trafodion objects in HBase:

```
>>get external hbase objects;

obj1
obj2

--- SQL operation complete.
>>
```

GET VERSION OF METADATA Statement

- "Considerations for GET VERSION OF METADATA"
- "Examples of GET VERSION OF METADATA"

The GET VERSION OF METADATA statement displays the version of the metadata in the Trafodion instance and indicates if the metadata is current.

GET VERSION OF METADATA is a Trafodion SQL extension.

```
GET VERSION OF METADATA
```

Considerations for GET VERSION OF METADATA

- If the metadata is compatible with the installed Trafodion software version, the GET VERSION OF METADATA statement indicates that the metadata is current:

  ```
  Current Version 3.0. Expected Version 3.0.
  Metadata is current.
  ```

- If the metadata is incompatible with the installed Trafodion software version, the GET VERSION OF METADATA statement indicates that you need to upgrade or reinitialize the metadata:

  ```
  Current Version 2.3. Expected Version 3.0.
    Metadata need to be upgraded or reinitialized.
  ```

Examples of GET VERSION OF METADATA

- This GET VERSION OF METADATA statement displays the metadata version in a Trafodion Release 1.0.0 instance:

  ```
  get version of metadata;

    Current Version 3.0. Expected Version 3.0.
    Metadata is current.

  --- SQL operation complete.
  ```

- This GET VERSION OF METADATA statement displays the metadata version in a Trafodion Release 0.9.0 instance:

  ```
  get version of metadata;

    Current Version 2.3. Expected Version 2.3.
    Metadata is current.

  --- SQL operation complete.
  ```

- If the metadata is incompatible with the installed Trafodion software version, you will see this output indicating that you need to upgrade or reinitialize the metadata:

  ```
  get version of metadata;

    Current Version 2.3. Expected Version 3.0.
    Metadata need to be upgraded or reinitialized.

  --- SQL operation complete.
  ```

GET VERSION OF SOFTWARE Statement

- "Considerations for GET VERSION OF SOFTWARE"
- "Examples of GET VERSION OF SOFTWARE"

The GET VERSION OF SOFTWARE statement displays the version of the Trafodion software that is installed on the system and indicates if it is current.

GET VERSION OF SOFTWARE is a Trafodion SQL extension.

```
GET VERSION OF SOFTWARE
```

Considerations for GET VERSION OF SOFTWARE

- If the software on the system is current, the GET VERSION OF SOFTWARE statement displays this output:

```
System Version 1.0.0. Expected Version 1.0.0.
  Software is current.
```

- In rare circumstances where something went wrong with the Trafodion software installation and mismatched objects were installed, the GET VERSION OF SOFTWARE statement displays this output:

```
System Version 0.9.1. Expected Version 1.0.0.
  Version of software being used is not compatible with version of software on the system.
```

Examples of GET VERSION OF SOFTWARE

- This GET VERSION OF SOFTWARE statement displays the software version for Trafodion Release 1.0.0:

```
get version of software;

  System Version 1.0.0. Expected Version 1.0.0.
  Software is current.

--- SQL operation complete.
```

- This GET VERSION OF SOFTWARE statement displays the software version for Trafodion Release 0.9.0:

```
get version of software;

  System Version 0.9.0. Expected Version 0.9.0.
  Software is current.

--- SQL operation complete.
```

- If something went wrong with the Trafodion software installation and if mismatched objects were installed, you will see this output indicating that the software being used is incompatible with the software on the system:

```
get version of software;

  System Version 0.9.1. Expected Version 1.0.0.
  Version of software being used is not compatible with version of software on the system.

--- SQL operation complete.
```

GRANT Statement

- "Syntax Description of GRANT"
- "Considerations for GRANT"
- "Examples of GRANT"

The GRANT statement grants access privileges on an SQL object to specified users or roles.

IMPORTANT: This statement works only when authentication and authorization are enabled in Trafodion. For more information, see Enabling Security Features in the Trafodion wiki.

```
GRANT {privilege [,privilege]... |ALL [PRIVILEGES]}
  ON [object-type] [schema.]object
  TO {grantee [,grantee ]...}
    [WITH GRANT OPTION]
    [[GRANTED] BY grantor]

privilege is:
   SELECT
  | DELETE
  | INSERT
  | REFERENCES
  | UPDATE
  | EXECUTE
  | USAGE

object-type is:
   TABLE
  | PROCEDURE
  | LIBRARY
  | FUNCTION

grantee is:
     auth-name

grantor is:
     role-name
```

Syntax Description of GRANT

privilege [,*privilege*] ... | ALL [PRIVILEGES]

specifies the privileges to grant. You can specify these privileges for an object.

SELECT	Can use the SELECT statement.
DELETE	Can use the DELETE statement.
INSERT	Can use the INSERT statement.
REFERENCES	Can create constraints that reference the object.
UPDATE	Can use the UPDATE statement on table objects.
EXECUTE	Can execute a stored procedure using a CALL statement or can execute a user-defined function (UDF).
USAGE	Can access a library using the CREATE PROCEDURE or CREATE FUNCTION statement. This privilege provides you with read access to the library's underlying library file.
ALL	All the applicable privileges. When you specify ALL for a table or view, this includes the SELECT, DELETE, INSERT, REFERENCES, and UPDATE privileges. When the object is a stored procedure or user-defined function (UDF), only the EXECUTE privilege is applied. When the object is a library, only the UPDATE and USAGE privileges are applied.

ON [*object-type*] [*schema*.]*object*

specifies an object on which to grant privileges. *object-type* can be:

- [TABLE] [*schema*.]*object*, where *object* is a table or view. See "Database Object Names" (page 198).
- [PROCEDURE] [*schema*.]*procedure-name*, where *procedure-name* is the name of a stored procedure in Java (SPJ) registered in the database.
- [LIBRARY] [*schema*.]*library-name*, where *library-name* is the name of a library object in the database.
- [FUNCTION] [*schema*.]*function-name*, where *function-name* is the name of a user-defined function (UDF) in the database.

TO {*grantee* [, *grantee*] ... }

specifies one or more *auth-names* to which you grant privileges.

auth-name

specifies the name of an authorization ID to which you grant privileges. See "Authorization IDs" (page 193). The authorization ID must be a registered database username, an existing role name, or PUBLIC. The name is a regular or delimited case-insensitive identifier. See "Case-Insensitive Delimited Identifiers" (page 221). If you grant a privilege to PUBLIC, the privilege remains available to all users, unless it is later revoked from PUBLIC.

WITH GRANT OPTION

specifies that the *auth-name* to which a privilege is granted may in turn grant the same privilege to other users or roles.

[GRANTED] BY *grantor*

allows you to grant privileges on behalf of a role. If not specified, the privileges will be granted on your behalf as the current user/grantor.

role-name

specifies a role on whose behalf the GRANT operation is performed. To grant the privileges on behalf of a role, you must be a member of the role, and the role must have the authority to grant the privileges; that is, the role must have been granted the privileges WITH GRANT OPTION.

Considerations for GRANT

Authorization and Availability Requirements

To grant a privilege on an object, you must have both that privilege and the right to grant that privilege. Privileges can be granted directly to you or to one of the roles you have been granted. You can grant a privilege on an object if you are the owner of the object (by which you are implicitly granted all privileges on the object) or the owner of the schema containing the object, or if you have been granted both the privilege and the WITH GRANT OPTION for the privilege.

If granting privileges on behalf of a role, you must specify the role in the [GRANTED] BY clause. To grant the privileges on behalf of a role, you must be a member of the role, and the role must have the authority to grant the privileges; that is, the role must have been granted the privileges WITH GRANT OPTION.

If you lack authority to grant one or more of the specified privileges, SQL returns a warning (yet does grant the specified privileges for which you do have authority to grant). If you lack authority to grant any of the specified privileges, SQL returns an error.

Examples of GRANT

- To grant SELECT and DELETE privileges on a table to two specified users:

```
GRANT SELECT, DELETE ON TABLE invent.partloc
  TO ajones, "MO.Neill@company.com";
```

- To grant SELECT privileges on a table to a user:

```
GRANT SELECT ON TABLE invent.partloc TO ajones;
```

GRANT COMPONENT PRIVILEGE Statement

- "Syntax Description of GRANT COMPONENT PRIVILEGE"
- "Considerations for GRANT COMPONENT PRIVILEGE"
- "Example of GRANT COMPONENT PRIVILEGE"

The GRANT COMPONENT PRIVILEGE statement grants one or more component privileges to a user or role. See "Privileges" (page 247) and "Roles" (page 248).

GRANT COMPONENT PRIVILEGE is a Trafodion SQL extension.

IMPORTANT: This statement works only when authentication and authorization are enabled in Trafodion. For more information, see Enabling Security Features in the Trafodion wiki.

```
GRANT COMPONENT PRIVILEGE {privilege-name [, privilege-name]...}
 ON component-name
 TO grantee
 [WITH GRANT OPTION] [[GRANTED] BY grantor]

grantee is:
     auth-name

grantor is:
     role-name
```

Syntax Description of GRANT COMPONENT PRIVILEGE

privilege-name

specifies one or more component privileges to grant. The comma-separated list can include only privileges within the same component.

Component	Component Privilege	Description
SQL_OPERATIONS	ALTER	Privilege to alter database objects
	ALTER_LIBRARY	Privilege to alter libraries
	ALTER_TABLE	Privilege to alter tables
	ALTER_VIEW	Privilege to alter views
	CREATE	Privilege to create database objects
	CREATE_CATALOG	Privilege to create catalogs in the database
	CREATE_INDEX	Privilege to create indexes
	CREATE_LIBRARY	Privilege to create libraries in the database
	CREATE_ROUTINE	Privilege to create stored procedures in Java (SPJs), user-defined functions (UDFs), table-mapping functions, and other routines in the database
	CREATE_SCHEMA	Privilege to create schemas in the database
	CREATE_TABLE	Privilege to create tables in the database
	CREATE_VIEW	Privilege to create views in the database
	DROP	Privilege to drop database objects

Component	Component Privilege	Description
	DROP_CATALOG	Privilege to drop catalogs
	DROP_INDEX	Privilege to drop indexes
	DROP_LIBRARY	Privilege to drop libraries
	DROP_ROUTINE	Privilege to drop stored procedures in Java (SPJs), user-defined functions (UDFs), table-mapping functions, and other routines from the database
	DROP_SCHEMA	Privilege to drop schemas
	DROP_TABLE	Privilege to drop tables
	DROP_VIEW	Privilege to drop views
	MANAGE_LIBRARY	Privilege to perform library-related commands, such as creating and dropping libraries
	MANAGE_LOAD	Privilege to perform LOAD and UNLOAD commands
	MANAGE_ROLES	Privilege to create, alter, drop, grant, and revoke roles
	MANAGE_STATISTICS	Privilege to update and display statistics
	MANAGE_USERS	Privilege to register or unregister users, alter users, and grant or revoke component privileges.
	QUERY_CANCEL	Privilege to cancel an executing query.
	SHOW	Privilege to run EXPLAIN, GET, INVOKE, and SHOW commands. The SHOW privilege has been granted to PUBLIC by default.

ON *component-name*

specifies a component name on which to grant component privileges. Currently, the only valid component name is SQL_OPERATIONS.

TO *grantee*

specifies an *auth-name* to which you grant component privileges.

auth-name

specifies the name of an authorization ID to which you grant privileges. See "Authorization IDs" (page 193). The authorization ID must be a registered database username, existing role name, or PUBLIC. The name is a regular or delimited case-insensitive identifier. See "Case-Insensitive Delimited Identifiers" (page 221). If you grant a privilege to PUBLIC, the privilege remains available to all users, unless it is later revoked from PUBLIC.

WITH GRANT OPTION

specifies that the *auth-name* to which a component privilege is granted may in turn grant the same component privilege to other users or roles.

[GRANTED] BY *grantor*

allows you to grant component privileges on behalf of a role. If not specified, the privileges will be granted on your behalf as the current user/grantor.

role-name

specifies a role on whose behalf the GRANT COMPONENT PRIVILEGE operation is performed. To grant the privileges on behalf of a role, you must be a member of the role, and the role must have the authority to grant the privileges; that is, the role must have been granted the privileges WITH GRANT OPTION.

Considerations for GRANT COMPONENT PRIVILEGE

- A user or role granted a component privilege WITH GRANT OPTION can grant the same component privilege to other users or roles.
- If all of the component privileges have already been granted, SQL returns an error.
- If one or more component privileges has already been granted, SQL silently ignores the granted privileges and proceeds with the grant operation.

Authorization and Availability Requirements

To grant a component privilege, you must have one of these privileges:

- User administrative privileges (that is, a user who has been granted the MANAGE_USERS component privilege). Initially, DB__ROOT is the only database user who has been granted the MANAGE_USERS component privilege.
- A user other than a user administrator who has the WITH GRANT OPTION for the component privilege.
- A user who was granted a role that has the WITH GRANT OPTION privilege for the component privilege.

Example of GRANT COMPONENT PRIVILEGE

Grant a component privilege, CREATE_TABLE, on a component, SQL_OPERATIONS, to SQLUSER1:

```
GRANT COMPONENT PRIVILEGE CREATE_TABLE ON SQL_OPERATIONS TO sqluser1;
```

GRANT ROLE Statement

- "Syntax Description of GRANT ROLE"
- "Considerations for GRANT ROLE"
- "Example of GRANT ROLE"

The GRANT ROLE statement grants one or more roles to a user. See "Roles" (page 248).

IMPORTANT: This statement works only when authentication and authorization are enabled in Trafodion. For more information, see Enabling Security Features in the Trafodion wiki.

```
GRANT ROLE {role-name [,role-name ]...}
  TO grantee

grantee is:
     database-username
```

Syntax Description of GRANT ROLE

role-name [,*role-name*] ...

specifies the existing roles to grant.

TO *grantee*

specifies the registered database username to whom to grant the roles.

Considerations for GRANT ROLE

- To grant roles to other grantees, you must own the roles or have user administrative privileges for the roles. You have user administrative privileges for roles if you have been granted the MANAGE_ROLES component privilege. Initially, DB__ROOT is the only database user who has been granted the MANAGE_ROLES component privilege.
- In Trafodion Release 0.9, when you grant a role to a grantee, the effects on any active sessions for the grantee are undefined, and users will need to disconnect such sessions and reconnect to establish a new session with the updated set of privileges. Starting in Trafodion Release 1.0, when you grant a role to a user, the additional privileges are automatically propagated to and detected by active sessions. There is no need for users to disconnect from and reconnect to a session to see the updated set of privileges.
- If any errors occur in processing a GRANT ROLE statement that names multiple roles, then no grants are done.
- If you attempt to grant a role but a grant with the same role and grantee already exists, SQL ignores the request and returns a successful operation.

Example of GRANT ROLE

To grant multiple roles to a grantee:

```
GRANT ROLE clerks, sales TO jsmith;
```

INSERT Statement

- "Syntax Description of INSERT"
- "Considerations for INSERT"
- "Examples of INSERT"

The INSERT statement is a DML statement that inserts rows in a table or view.

```
INSERT INTO table [(target-col-list)] insert-source

target-col-list is:
   colname [,colname]...

insert-source is:
   query-expr [order-by-clause] [access-clause] | DEFAULT VALUES
```

Syntax Description of INSERT

table

names the user table or view in which to insert rows. *table* must be a base table or an updatable view.

(target-col-list)

names the columns in the table or view in which to insert values. The data type of each target column must be compatible with the data type of its corresponding source value. Within the list, each target column must have the same position as its associated source value, whose position is determined by the columns in the table derived from the evaluation of the query expression (*query-expr*).

If you do not specify all of the columns in *table* in the *target-col-list*, column default values are inserted into the columns that do not appear in the list. See "Column Default Settings" (page 194).

If you do not specify *target-col-list*, row values from the source table are inserted into all columns in *table*. The order of the column values in the source table must be the same order as that of the columns specified in the CREATE TABLE for *table*. (This order is the same as that of the columns listed in the result table of SHOWDDL *table*.)

insert-source

specifies the rows of values to be inserted into all columns of *table* or, optionally, into specified columns of *table*.

query-expr

For the description of *query-expr*, *order-by-clause*, and *access-clause*, see the "SELECT Statement" (page 138).

DEFAULT VALUES

is equivalent to a *query-expr* of the form VALUES (DEFAULT, ...). The value of each DEFAULT is the default value defined in the column descriptor of *colname*, which is contained in the table descriptor of *table*. Each default value is inserted into its column to form a new row. If you specify DEFAULT VALUES, you cannot specify a column list. You can use DEFAULT VALUES only when all columns in *table* have default values.

Considerations for INSERT

Authorization Requirements

INSERT requires authority to read and write to the table or view receiving the data and authority to read tables or views specified in the query expression (or any of its subqueries) in the INSERT statement.

Transaction Initiation and Termination

The INSERT statement automatically initiates a transaction if no transaction is active. Alternatively, you can explicitly initiate a transaction with the BEGIN WORK statement. After a transaction is started, the SQL statements execute within that transaction until a COMMIT or ROLLBACK is encountered or an error occurs. If AUTOCOMMIT is ON, the transaction terminates at the end of the INSERT statement.

Self-Referencing INSERT and BEGIN WORK or AUTOCOMMIT OFF

A self-referencing INSERT statement is one that references, in the statement's *insert-source*, the same table or view into which rows will be inserted (see "Examples of Self-Referencing Inserts" (page 121)). A self-referencing INSERT statement will not execute correctly and an error is raised if either BEGIN WORK or AUTOCOMMIT OFF is used unless the compiler's plan sorts the rows before they are inserted. If you want to use a self-referencing INSERT statement, you should avoid the use of BEGIN WORK or AUTOCOMMIT OFF. For information about AUTOCOMMIT, see the "SET TRANSACTION Statement" (page 157).

Isolation Levels of Transactions and Access Options of Statements

The isolation level of an SQL transaction defines the degree to which the operations on data within that transaction are affected by operations of concurrent transactions. When you specify access options for the DML statements within a transaction, you override the isolation level of the containing transaction. Each statement then executes with its individual access option.

Use of a VALUES Clause for the Source Query Expression

If the query expression consists of the VALUES keyword followed by rows of values, each row consists of a list of value expressions or a row subquery (a subquery that returns a single row of column values). A value in a row can also be a scalar subquery (a subquery that returns a single row consisting of a single column value).

Within a VALUES clause, the operands of a value expression can be numeric, string, datetime, or interval values; however, an operand cannot reference a column (except in the case of a scalar or row subquery returning a value or values in its result table).

Requirements for Inserted Rows

Each row to be inserted must satisfy the constraints of the table or underlying base table of the view. A table constraint is satisfied if the check condition is not false—it is either true or has an unknown value.

Using Compatible Data Types

To insert a row, you must provide a value for each column in the table that has no default value. The data types of the values in each row to be inserted must be compatible with the data types of the corresponding target columns.

Inserting Character Values

Any character string data type is compatible with all other character string data types that have the same character set. For fixed length, an inserted value shorter than the column length is padded on the right with blank characters of the appropriate character set (for example, ISO88591 blanks (HEX20). If the value is longer than the column length, string truncation of nonblank trailing characters returns an error, and the truncated string is not inserted.

For variable length, a shorter inserted value is not padded. As is the case for fixed length, if the value is longer than the column length, string truncation of nonblank trailing characters returns an error, and the truncated string is not inserted.

Inserting Numeric Values

Any numeric data type is compatible with all other numeric data types. If you insert a value into a numeric column that is not large enough, an overflow error occurs. If a value has more digits to the right of the decimal point than specified by the scale for the column definition, the value is truncated.

Inserting Interval Values

A value of interval data type is compatible with another value of interval data type only if the two data types are both year-month or both day-time intervals.

Inserting Date and Time Values

Date, time, and timestamp are the three Trafodion SQL datetime data types. A value with a datetime data type is compatible with another value with a datetime data type only if the values have the same datetime fields.

Inserting Nulls

and inserting values with specific data types, you might want to insert nulls. To insert null, use the keyword NULL. NULL only works with the VALUES clause. Use `cast (null as type)` for select-list.

Examples of INSERT

- Insert a row into the CUSTOMER table without using a *target-col-list*:

```
INSERT INTO sales.customer
  VALUES (4777, 'ZYROTECHNIKS', '11211 40TH ST.',
          'BURLINGTON', 'MASS.', '01803', 'A2');

--- 1 row(s) inserted.
```

  The column name list is not specified for this INSERT statement. This operation works because the number of values listed in the VALUES clause is equal to the number of columns in the CUSTOMER table, and the listed values appear in the same order as the columns specified in the CREATE TABLE statement for the CUSTOMER table.

  By issuing this SELECT statement, this specific order is displayed:

```
SELECT * FROM sales.customer
  WHERE custnum = 4777;

CUSTNUM  CUSTNAME       STREET          ... POSTCODE  CREDIT
-------  -------------  --------------      --------  ------
  4777   ZYROTECHNIKS   11211 4OTH ST. ... 01803     A2

--- 1 row(s) selected.
```

- Insert a row into the CUSTOMER table using a *target-col-list*:

```
INSERT INTO sales.customer
  (custnum, custname, street, city, state, postcode)
  VALUES (1120, 'EXPERT MAILERS', '5769 N. 25TH PL',
          'PHOENIX', 'ARIZONA', '85016');

--- 1 row(s) inserted.
```

  Unlike the previous example, the insert source of this statement does not contain a value for the CREDIT column, which has a default value. As a result, this INSERT must include the column name list.

  This SELECT statement shows the default value 'C1' for CREDIT:

```
SELECT * FROM sales.customer
  WHERE custnum = 1120;

CUSTNUM  CUSTNAME        STREET                POSTCODE  CREDIT
-------  --------------  ------------------    --------  ------
  1120   EXPERT MAILERS  5769 N. 25TH PL       85016     C1
```

```
--- 1 row(s) selected.
```

- Insert multiple rows into the JOB table by using only one INSERT statement:

```
INSERT INTO persnl.job
   VALUES (100,'MANAGER'),
          (200,'PRODUCTION SUPV'),
          (250,'ASSEMBLER'),
          (300,'SALESREP'),
          (400,'SYSTEM ANALYST'),
          (420,'ENGINEER'),
          (450,'PROGRAMMER'),
          (500,'ACCOUNTANT'),
          (600,'ADMINISTRATOR'),
          (900,'SECRETARY');

--- 10 row(s) inserted.
```

- The PROJECT table consists of five columns using the data types numeric, varchar, date, timestamp, and interval. Insert values by using these types:

```
INSERT INTO persnl.project
  VALUES (1000, 'SALT LAKE CITY', DATE '2007-10-02',
  TIMESTAMP '2007-12-21 08:15:00.00', INTERVAL '30' DAY);

--- 1 row(s) inserted.
```

- Suppose that CUSTLIST is a view of all columns of the CUSTOMER table except the credit rating. Insert information from the SUPPLIER table into the CUSTOMER table through the CUSTLIST view, and then update the credit rating:

```
INSERT INTO sales.custlist
  (SELECT * FROM invent.supplier
   WHERE suppnum = 10);

UPDATE sales.customer
  SET credit = 'A4'
  WHERE custnum = 10;
```

  You could use this sequence in the following situation. Suppose that one of your suppliers has become a customer. If you use the same number for both the customer and supplier numbers, you can select the information from the SUPPLIER table for the new customer and insert it into the CUSTOMER table through the CUSTLIST view (as shown in the example).

  This operation works because the columns of the SUPPLIER table contain values that correspond to the columns of the CUSTLIST view. Further, the credit rating column in the CUSTOMER table is specified with a default value. If you want a credit rating that is different from the default, you must update this column in the row of new customer data.

Examples of Self-Referencing Inserts

- This is an example of a self-referencing insert:

```
insert into table1 select pk+?, b, c from table1
```

- This is an example of a self-referencing insert where the target of the insert, `table1`, is also used in a subquery of the insert-source:

```
insert into table1
  select a+16, b, c from table2 where table2.b not in
    (select b from table1 where a > 16)
```

  The source table is not affected by the insert.

INVOKE Statement

- "Syntax Description of INVOKE "
- "Considerations for INVOKE"
- "Example of INVOKE"

The INVOKE statement generates a record description that corresponds to a row in the specified table, view, or index. The record description includes a data item for each column in the table, view, or index, including the primary key but excluding the SYSKEY column. It includes the SYSKEY column of a view only if the view explicitly listed the column in its definition.

INVOKE is a Trafodion SQL extension.

```
INVOKE  table-name
```

Syntax Description of INVOKE

table-name

specifies the name of a table, view, or index for which to generate a record description. See "Database Object Names" (page 198).

Considerations for INVOKE

Required Privileges

To issue an INVOKE statement, one of the following must be true:

- You are DB__ROOT.
- You are the owner of the table.
- You have the SHOW component privilege for the SQL_OPERATIONS component. The SHOW component privilege is granted to PUBLIC by default.
- You have the SELECT privilege on the target table.

Example of INVOKE

This command generates a record description of the table T:

```
SQL>invoke trafodion.seabase.t;


-- Definition of Trafodion table TRAFODION.SEABASE.T
-- Definition current  Wed Mar  5 10:36:06 2014

  (
    A                                INT NO DEFAULT NOT NULL NOT DROPPABLE
  )
  PRIMARY KEY (A ASC)

--- SQL operation complete.
```

MERGE Statement

- "Syntax Description of MERGE "
- "Considerations for MERGE "
- "Example of MERGE "

The MERGE statement:

- Updates a table if the row exists or inserts into a table if the row does not exist. This is upsert functionality.
- Updates (merges) matching rows from one table to another.

```
MERGE INTO table [using-clause]
   on-clause
   {[when-matched-clause]|[when-not-matched-clause]} ...

using-clause is:
   USING (select-query) AS derived-table-name [derived-column-names]

on-clause is:
   ON predicate

when-matched-clause is:
   WHEN MATCHED THEN UPDATE SET set-clause [WHERE predicate]
   WHEN MATCHED THEN DELETE

when-not-matched-clause is:
   WHEN NOT MATCHED THEN INSERT insert-values-list

insert-values-list is:
   [(column1, ..., columnN )] VALUES (value1, ..., valueN)
```

Syntax Description of MERGE

table

is the ANSI logical name for the table.

ON *predicate*

used to determine if a row is or is not present in the table. The ON predicate must be a predicate on the clustering key of the table if the MERGE has a *when-not-matched-clause*. The clustering key can be a single or multi-column key.

The ON predicate must select a unique row if the MERGE has a *when-not-matched-clause*.

Considerations for MERGE

Upsert Using Single Row

A MERGE statement allows you to specify a set of column values that should be updated if the row is found, and another row to be inserted if the row is not found. The ON predicate must select exactly one row that is to be updated if the MERGE statement has an INSERT clause.

In a MERGE statement, at least one of the clauses *when-matched* or *when-not-matched* must be specified. Note the following:

- If a *when-matched* clause is present and the WHERE predicate in the UPDATE is satisfied, the columns in the SET clause are updated.
- If a *when-matched* clause is present and the WHERE predicate in the UPDATE is not satisfied, the columns in the SET clause are not updated.

- If a *when-matched* clause is present and the UPDATE has no WHERE predicate, the columns in the SET clause are updated.
- If a *when-not-matched* clause is present and columns are explicitly specified in the INSERT clause, the specified values for those columns are inserted. Missing columns are updated using the default values for those columns.

This example updates column b to 20 if the row with key column a with value 10 is found. A new row (10, 30) is inserted if the row is not found in table t.

```
MERGE INTO t ON a = 10
 WHEN MATCHED THEN UPDATE SET b = 20
 WHEN NOT MATCHED THEN INSERT VALUES (10, 30)
```

This example updates column b to 20 if column a with value 10 is found. If column a with value 10 is not found, nothing is done.

```
MERGE INTO t ON a = 10
 WHEN MATCHED THEN UPDATE SET b = 20
```

This example inserts values (10, 30) if column a with value 10 is not found. If column a with value 10 is found, nothing is done.

```
MERGE INTO t ON a = 10
 WHEN NOT MATCHED THEN INSERT VALUES (10, 30)
```

Conditional Upsert Using Single Row

In this example, the MERGE statement uses a single-row conditional upsert that inserts one row (keycol, col, seqnum) value if a row with that keycol (parameter-specified) value is not yet in table d. Otherwise, the MERGE statement updates that row's col and seqnum columns if that row's seqnum is higher than the current (parameter-specified) sequence number. If the matching row's seqnum column value is not higher than the current sequence number, then that matched row is not updated.

```
MERGE INTO d ON keycol = ?
  WHEN MATCHED THEN UPDATE SET (col, seqnum) = (?, ?) WHERE seqnum < ?
  WHEN NOT MATCHED THEN INSERT (keycol, col, seqnum) VALUES (?, ?, ?)
```

The optional WHERE predicate in the *when-matched-then-update* clause is useful when the update is wanted only if the given condition is satisfied. Consider this use case. Suppose object X is represented as a row in table T. Also, suppose a stream of updates exists for object X. The updates are marked by a sequence number at their source. However, the updates flow through a network which does not guarantee first-in, first-out delivery. In fact, the updates may arrive out-of-order to the database. In this case, the last update (the one with the current highest sequence number) should always win in the database. The MERGE statement shown above can be used to satisfy this use case:

- A stream of updates for table d exists that are sequenced by a sequence number seqnum at their source
- The updates race through the network and may arrive in the database in any order, and
- You want to guarantee that the last update (the one with the highest seqnum) always wins in the database.

Restrictions

- The MERGE statement does not use ESP parallelism.
- A merged table cannot be a view.
- Merge is not allowed if the table has constraints.
- The *on-clause* cannot contain a subquery. This statement is not allowed:

  ```
  MERGE INTO t ON a = (SELECT a FROM t1) WHEN ...
  ```

- The optional WHERE predicate in the when-matched clause cannot contain a subquery or an aggregate function. These statements are not allowed:

```
MERGE INTO t ON a = 10
WHEN MATCHED THEN UPDATE SET b=4 WHERE b=(SELECT b FROM t1)
WHEN NOT MATCHED THEN INSERT VALUES (10,30);

MERGE INTO t ON a=10
WHEN MATCHED THEN UPDATE SET b=4 WHERE b=MAX(b)
WHEN NOT MATCHED THEN INSERT VALUES (10,30);
```

- The UPDATE SET clause in a MERGE statement cannot contain a subquery. This statement is not allowed:

```
MERGE INTO t ON a = 1 WHEN MATCHED THEN UPDATE SET b = (SELECT a FROM t1)
```

- The *insert-values-list* clause in a MERGE statement cannot contain a subquery. This statement is not allowed:

```
MERGE INTO t ON a = 1 WHEN NOT MATCHED THEN INSERT VALUES ((SELECT a FROM t1))
```

- Use of a non-unique *on-clause* for a MERGE update is allowed only if no INSERT clause exists.

```
MERGE INTO t USING (SELECT a,b FROM t1) x ON t.a=x.a
   WHEN MATCHED THEN UPDATE SET b=x.b;
```

  In this example, `t.a=x.a` is not a fully qualified unique primary key predicate.

- Use of a non-unique *on-clause* for a MERGE delete is allowed only if no INSERT clause exists.

```
MERGE INTO t USING (SELECT a,b FROM t1) x ON t.a=x.a
   WHEN MATCHED THEN DELETE;
```

MERGE From One Table Into Another

The MERGE statement can be used to upsert all matching rows from the source table into the target table. Each row from the source table is treated as the source of a single upsert statement. The *using-clause* contains the *select-query* whose output is used as the source to the MERGE statement.

The source *select-query* must be renamed using the AS clause.

```
MERGE INTO t ON
    USING (select-query) AS Z(X) ON col = Z.X
    WHEN MATCHED THEN . . .
```

For each row selected out of the select-query, the MERGE statement is evaluated. Values selected are used in the *on-clause* to join with the column of the merged table. If the value is found, it is updated. If it is not found, the insert is done. The restrictions are the same as those for "Upsert Using Single Row" (page 123).

Example of MERGE

This query extracts derived columns `a` and `b` from the USING query as derived table `z` and use each row to join to the merged table `t` based on the *on-clause*. For each matched row, column `b` in table `t` is updated using column `b` in derived table `z`. For rows that are not matched, values `z.a` and `z.b` are inserted.

```
MERGE INTO t USING
  (SELECT * FROM t1) z(a,b) on a = z.a
  WHEN MATCHED THEN UPDATE SET b = z.b
  WHEN NOT MATCHED THEN INSERT VALUES (z.a, z.b);
```

PREPARE Statement

- "Syntax Description of PREPARE"
- "Considerations for PREPARE"
- "Examples of PREPARE"

The PREPARE statement compiles an SQL statement for later use with the EXECUTE statement in the same Trafodion Command Interface (TrafCI) session.

You can also use PREPARE to check the syntax of a statement without executing the statement in the same TrafCI session.

```
PREPARE statement-name FROM statement
```

Syntax Description of PREPARE

statement-name

is an SQL identifier that specifies a name to be used for the prepared statement. See "Identifiers" (page 221). The statement name should be a character string and not a numeric value. If you specify the name of an existing prepared statement, the new statement overwrites the previous one.

statement

specifies the SQL statement to prepare.

Considerations for PREPARE

Availability of a Prepared Statement

If a PREPARE statement fails, any subsequent attempt to run EXECUTE on the named statement fails.

Only the TrafCI session that executes the PREPARE can run EXECUTE on the prepared statement. The prepared statement is available for running EXECUTE until you terminate the TrafCI session.

A statement must be compiled by PREPARE before you can run EXECUTE on it. However, after the statement is compiled, you can run EXECUTE on the statement multiple times without recompiling the statement.

Examples of PREPARE

- Prepare a SELECT statement, checking for syntax errors:

```
SQL>prepare empsal from
+>select salary from employee
+>where jobcode = 100;

*** ERROR[4082] Table, view or stored procedure NEO.INVENT.EMPLOYEE does not exist or is inaccessible.
*** ERROR[8822] The statement was not prepared.

SQL>
```

- Prepare a SELECT statement with an unnamed parameter (?) and later run EXECUTE on it:

```
SQL>prepare findsal from
+>select salary from persnl.employee
+>where jobcode = ?;

--- SQL command prepared.

SQL>execute findsal using 450;

SALARY
----------
  32000.00
  33000.50
  40000.00
```

```
  32000.00
  45000.00

--- 5 row(s) selected.

SQL>
```

- Prepare a SELECT statement with a named parameter (*?param-name*) and later run EXECUTE on it:

```
SQL>prepare findsal from
+>select salary from persnl.employee
+>where jobcode = ?job;

--- SQL command prepared.

SQL>set param ?job 450

SQL>execute findsal;
SALARY
----------
  32000.00
  33000.50
  40000.00
  32000.00
  45000.00

--- 5 row(s) selected.

SQL>
```

For more information, see the "EXECUTE Statement" (page 98).

REGISTER USER Statement

- "Syntax Description of REGISTER USER"
- "Considerations for REGISTER USER"
- "Examples of REGISTER USER"

The REGISTER USER statement registers a user in the SQL database, associating the user's login name with a database username.

REGISTER USER is a Trafodion SQL extension.

NOTE: The user's login name is also the name by which the user is defined in the directory service, so the syntax description below refers to it as the *directory-service username*.

```
REGISTER USER directory-service-username [ AS database-username ]
```

Syntax Description of REGISTER USER

directory-service-username

is the name that identifies the user in the directory service. This is also the name the user specifies when logging in to a Trafodion database. The *directory-service-username* is a regular or delimited case-insensitive identifier. See "Case-Insensitive Delimited Identifiers" (page 221).

database-username

is a regular or delimited case-insensitive identifier that denotes the username as defined in the database. The database username cannot be identical to a registered database username or an existing role name. However, it can be the same as the directory-service username. If you omit the AS *database-username* clause, the database username will be the same as the directory-service username.

Considerations for REGISTER USER

Who Can Register a User

To register a user, you must have user administrative privileges. You have user administrative privileges if you have been granted the MANAGE_USERS component privilege. Initially, DB__ROOT is the only database user who has been granted the MANAGE_USERS component privilege.

Add the User to the Directory Before Registering the User

Add the user to the appropriate directory service before you register the user. Otherwise, REGISTER USER will fail.

AS *database-username* Clause

Use the AS *database-username* clause to assign a database username that is different than the username defined in the directory service. In particular, it is often convenient to assign a database username that is shorter and easier to type than the directory-service username. For example, if the user logs on as John.Allen.Doe.the.Second@mycompany.com, you might want to assign the user a database username of JDoe.

Database usernames are authorization IDs. If you specify a name already assigned to another user or to an existing role, the command will fail. For more information, see "Authorization IDs" (page 193).

Reserved Names

PUBLIC, _SYSTEM, NONE, and database usernames beginning with DB__ are reserved. You cannot register users with any such name.

Username Length

Database usernames are limited to 128 characters.

Examples of REGISTER USER

- To register a user and assign a database username different than the user's login name:

```
REGISTER USER "jsmith@hp.com" AS jsmith;
```

- To register a user without specifying a database username, so the database username will be the same as the user's login name:

```
REGISTER USER "jsmith@hp.com";
```

REVOKE Statement

- "Syntax Description of REVOKE"
- "Considerations for REVOKE"
- "Examples of REVOKE"

The REVOKE statement revokes access privileges on an SQL object from specified users or roles.

IMPORTANT: This statement works only when authentication and authorization are enabled in Trafodion. For more information, see Enabling Security Features in the Trafodion wiki.

```
REVOKE [GRANT OPTION FOR]
   {privilege [,privilege]...| ALL [PRIVILEGES]}
   ON [object-type] [schema.]object
   FROM {grantee [,grantee]...}
   [[GRANTED] BY grantor]
   [RESTRICT | CASCADE]

privilege is:
   SELECT
  | DELETE
  | INSERT
  | REFERENCES
  | UPDATE
  | EXECUTE
  | USAGE

object-type is:
   TABLE
  | PROCEDURE
  | LIBRARY
  | FUNCTION

grantee is:
     auth-name

grantor is:
     role-name
```

Syntax Description of REVOKE

GRANT OPTION FOR

specifies that the grantee's authority to grant the specified privileges to other users or roles (that is, WITH GRANT OPTION) be revoked. This is an optional clause. When this clause is specified, only the ability to grant the privilege to another user is revoked.

privilege [, *privilege*] ... | ALL [PRIVILEGES]

specifies the privileges to revoke. You can specify these privileges for an object:

SELECT	Revokes the ability to use the SELECT statement.
DELETE	Revokes the ability to use the DELETE statement.
INSERT	Revokes the ability to use the INSERT statement.
REFERENCES	Revokes the ability to create constraints that reference the object.
UPDATE	Revokes the ability to use the UPDATE statement.
EXECUTE	Revokes the ability to execute a stored procedure using a CALL statement or revokes the ability to execute a user-defined function (UDF).
USAGE	Revokes the ability to access a library using the CREATE PROCEDURE or CREATE FUNCTION statement. Revokes read access to the library's underlying library file.
ALL	Revokes the ability to use all privileges that apply to the object type. When you specify ALL for a table or view, this includes the SELECT, DELETE, INSERT, REFERENCES, and UPDATE

privileges. When the object is a stored procedure or user-defined function (UDF), this includes the EXECUTE privilege. When the object is a library, this includes the UPDATE and USAGE privileges.

ON [*object-type*] [*schema*.]*object*

specifies an object on which to grant privileges. *object-type* can be:

- [TABLE] [*schema*.]*object*, where *object* is a table or view. See "Database Object Names" (page 198).
- [PROCEDURE] [*schema*.]*procedure-name*, where *procedure-name* is the name of a stored procedure in Java (SPJ) registered in the database. See "Database Object Names" (page 198).
- [LIBRARY] [*schema*.]*library-name*, where *library-name* is the name of a library object in the database. See "Database Object Names" (page 198).
- [FUNCTION] [*schema*.]*function-name*, where *function-name* is the name of a user-defined function in the database. See "Database Object Names" (page 198).

FROM {*grantee* [,*grantee*] ... }

specifies an *auth-name* from which you revoke privileges.

auth-name

specifies the name of an authorization ID from which you revoke privileges. See "Authorization IDs" (page 193). The authorization ID must be a registered database username, existing role name, or PUBLIC. The name is a regular or delimited case-insensitive identifier. See "Case-Insensitive Delimited Identifiers" (page 221).

[GRANTED] BY *grantor*

allows you to revoke privileges on behalf of a role. If not specified, the privileges will be revoked on your behalf as the current user/grantor.

role-name

specifies a role on whose behalf the GRANT operation was performed. To revoke the privileges on behalf of a role, you must be a member of the role, and the role must have the authority to revoke the privileges; that is, the role must have been granted the privileges WITH GRANT OPTION.

[RESTRICT | CASCADE]

If you specify RESTRICT, the REVOKE operation fails if any privileges were granted or any objects were created based upon the specified privileges.

If you specify CASCADE, any such dependent privileges and objects are removed as part of the REVOKE operation.

The default value is RESTRICT.

Considerations for REVOKE

Authorization and Availability Requirements

You can revoke privileges for which you are the grantor, either through a direct grant or a grant done on your behalf. If you are revoking privileges that were granted on behalf of a role, you must be a member of the role, and you must specify the role in the [GRANTED] BY clause.

If one or more privileges have not been granted, SQL returns a warning.

When you specify the CASCADE option, all objects that were created based upon the privileges being revoked are removed.

Examples of REVOKE

- To revoke the privilege to grant SELECT and DELETE privileges on a table from a user:

```
REVOKE GRANT OPTION FOR SELECT, DELETE ON TABLE invent.partloc
  FROM jsmith;
```

- To revoke the privilege to grant SELECT and DELETE privileges on a table from a user and a role:

```
REVOKE GRANT OPTION FOR SELECT, DELETE ON TABLE invent.partloc
  FROM jsmith, clerks;
```

- To revoke a user's SELECT privileges on a table:

```
-- User administrator grants the SELECT privilege to JSMITH:
GRANT SELECT ON TABLE invent.partloc TO jsmith
  WITH GRANT OPTION;
-- JSMITH grants the SELECT privilege to AJONES:
GRANT SELECT ON TABLE invent.partloc TO ajones;
-- If the user administrator attempts to revoke the SELECT
--   privilege from JSMITH, this would fail because
--   of the privilege granted to AJONES based on the
--   privilege granted to JSMITH.
-- To successfully revoke the SELECT privilege from
--   JSMITH, the SELECT privilege granted to AJONES
--   must be revoked first. For this example:
-- 1. JSMITH revokes the SELECT privilege granted to AJONES:
REVOKE SELECT ON TABLE invent.partloc FROM ajones;
-- 2. User administrator revokes the SELECT privilege on the
--   table from JSMITH:
REVOKE SELECT ON TABLE invent.partloc FROM jsmith RESTRICT;
-- The REVOKE operation succeeds.
-- An easier way to make the REVOKE operation successful is
--   to use the CASCADE option:
REVOKE SELECT ON TABLE invent.partloc FROM jsmith CASCADE;
-- The REVOKE operation succeeds because the CASCADE option
-- causes all specified privileges, and all privileges that
-- were granted based upon the specified privileges, to be
-- removed.
```

- Administration in the shipping department decides that the CLERKS role should no longer be able to grant privileges on the `invent.partloc` table. Fred has recently moved to another department, so JSMITH revokes the SELECT privilege on the `invent.partloc` table from Fred, who was granted the privilege by CLERKS. Then, JSMITH revokes the grant option from CLERKS:

```
REVOKE SELECT on table invent.partloc FROM fred
GRANTED BY clerks;

REVOKE GRANT OPTION FOR SELECT ON TABLE invent.partloc FROM
clerks;
```

REVOKE COMPONENT PRIVILEGE Statement

- "Syntax Description of REVOKE COMPONENT PRIVILEGE"
- "Considerations for REVOKE COMPONENT PRIVILEGE"
- "Example of REVOKE COMPONENT PRIVILEGE"

The REVOKE COMPONENT PRIVILEGE statement removes one or more component privileges from a user or role. See "Privileges" (page 247) and "Roles" (page 248).

REVOKE COMPONENT PRIVILEGE is a Trafodion SQL extension.

IMPORTANT: This statement works only when authentication and authorization are enabled in Trafodion. For more information, see Enabling Security Features in the Trafodion wiki.

```
REVOKE [GRANT OPTION FOR]
  COMPONENT PRIVILEGE {privilege-name[, privilege-name]...}
  ON component-name
  FROM grantee
  [[GRANTED] BY grantor]

grantee is:
     auth-name

grantor is:
     role-name
```

Syntax Description of REVOKE COMPONENT PRIVILEGE

GRANT OPTION FOR

specifies that the grantee's authority to grant the specified component privileges to other users or roles (that is, WITH GRANT OPTION) be revoked. This is an optional clause. When this clause is specified, only the ability to grant the component privilege to another user is revoked.

privilege-name

specifies one or more component privileges to revoke. The comma-separated list can include only privileges within the same component.

ON *component-name*

specifies a valid component name on which to revoke component privileges. Currently, the only valid component name is SQL_OPERATIONS.

FROM *grantee*

specifies an *auth-name* from which you revoke the component privileges.

auth-name

specifies the name of an authorization ID from which you revoke privileges. See "Authorization IDs" (page 193). The authorization ID must be a registered database username, existing role name, or PUBLIC. The name is a regular or delimited case-insensitive identifier. See "Case-Insensitive Delimited Identifiers" (page 221).

[GRANTED] BY *grantor*

allows you to revoke component privileges on behalf of a role. If not specified, the component privileges will be revoked on your behalf as the current user/grantor.

role-name

specifies a role on whose behalf the GRANT COMPONENT PRIVILEGE operation was performed. To revoke the privileges on behalf of a role, you must be a member of the role, and the role must have the authority to revoke the privileges; that is, the role must have been granted the privileges WITH GRANT OPTION.

Considerations for REVOKE COMPONENT PRIVILEGE

- At revoke time, all privileges granted WITH GRANT OPTION are removed. That is, the revoke behavior is CASCADE.
- If none of the component privileges has been granted, SQL returns an error.
- If one or more component privileges have not been granted, SQL silently ignores those privileges and proceeds with the revoke operation.
- Component privileges must be revoked before a role can be dropped or a user unregistered. If any privileges have been granted to a role or user, an error is returned when that role is dropped or the user unregistered. For more information, see the "DROP ROLE Statement" (page 93) and the "UNREGISTER USER Statement" (page 168).

Authorization and Availability Requirements

You can revoke component privileges for which you are the grantor, either through a direct grant or a grant done on your behalf. If you are revoking privileges that were granted on behalf of a role, you must be a member of the role, and you must specify the role in the [GRANTED] BY clause.

Example of REVOKE COMPONENT PRIVILEGE

Revoke a component privilege from SQLUSER1:

```
REVOKE COMPONENT PRIVILEGE CREATE_TABLE ON
SQL_OPERATIONS FROM sqluser1;
```

REVOKE ROLE Statement

- "Syntax Description of REVOKE ROLE"
- "Considerations for REVOKE ROLE"
- "Examples of REVOKE ROLE"

The REVOKE ROLE statement removes one or more roles from a user. See "Roles" (page 248).

IMPORTANT: This statement works only when authentication and authorization are enabled in Trafodion. For more information, see Enabling Security Features in the Trafodion wiki.

```
REVOKE ROLE {role-name [,role-name]...}
  FROM grantee
    [RESTRICT | CASCADE]

grantee is:
     database-username
```

Syntax Description of REVOKE ROLE

role-name [, *role-name*] ...

specifies the valid roles to revoke.

FROM *grantee*

specifies the registered database username from whom you revoke the roles.

[RESTRICT | CASCADE]

If you specify RESTRICT, the REVOKE ROLE operation fails if any privileges were granted to the role or any objects were created based upon those privileges.

If you specify CASCADE, any dependent privileges are removed as part of the REVOKE ROLE operation.

The default value is RESTRICT.

Considerations for REVOKE ROLE

- To revoke roles from users, you must own the roles or have user administrative privileges for the roles. You have user administrative privileges for roles if have been granted the MANAGE_ROLES component privilege. Initially, DB__ROOT is the only database user who has been granted the MANAGE_ROLES component privilege.
- If RESTRICT (or nothing) is specified and if you want to revoke a role from a user that has created objects based solely on role privileges, you must drop the objects before revoking the role. However, if you specify CASCADE, the dependent objects are automatically dropped, and the role is revoked.
- All of the specified roles must have been granted to the specified user. If any role has not been granted to the user, the operation returns an error, and no roles are revoked.
- In Trafodion Release 0.9, when you revoke a role from a user that has active sessions, you will need to disconnect the active sessions and reconnect for the reduction in privileges to take full effect. Starting in Trafodion Release 1.0, when you revoke a role from a user, the reduction in privileges is automatically propagated to and detected by active sessions. There is no need for users to disconnect from and reconnect to a session to see the updated set of privileges.
- If the REVOKE ROLE names multiple roles and any errors occur in processing, no revokes are performed.

Examples of REVOKE ROLE

- To revoke multiple roles from a user:

```
REVOKE ROLE clerks, sales FROM jsmith;
```

- To revoke a role with dependent objects from a user:

```
-- CMILLER grants a role to AJONES:
GRANT ROLE sales TO ajones;
-- CMILLER grants a privilege to the role:
GRANT SELECT ON TABLE invent.partloc TO sales;
-- AJONES creates a view based upon the privilege granted
--   to the role granted to him:
CREATE VIEW invent.partlocview (partnum, loc_code)
    AS SELECT partnum, loc_code FROM invent.partloc;
-- If CMILLER attempts to revoke the role from AJONES,
--   this would fail because of the view created based
--   upon the privilege granted to the role granted to
--   AJONES.
-- CMILLER revokes the role from AJONES with the CASCADE
   option:
REVOKE ROLE sales from AJONES CASCADE;
-- The REVOKE ROLE operation succeeds, and all dependent
   object privileges are revoked.
```

ROLLBACK WORK Statement

- "Syntax Description of ROLLBACK WORK"
- "Considerations for ROLLBACK WORK"
- "Example of ROLLBACK WORK"

The ROLLBACK WORK statement undoes all database modifications to objects made during the current transaction and ends the transaction. See "Transaction Management" (page 25).

Syntax Description of ROLLBACK WORK

```
ROLLBACK [WORK]
```

WORK is an optional keyword that has no effect.

ROLLBACK WORK issued outside of an active transaction generates error 8609.

Considerations for ROLLBACK WORK

Begin and End a Transaction

BEGIN WORK starts a transaction. COMMIT WORK or ROLLBACK WORK ends a transaction.

Example of ROLLBACK WORK

Suppose that you add an order for two parts numbered 4130 to the ORDERS and ODETAIL tables. When you update the PARTLOC table to decrement the quantity available, you discover no such part number exists in the given location.

Use ROLLBACK WORK to terminate the transaction without committing the database changes:

```
BEGIN WORK;

INSERT INTO sales.orders
  VALUES (124, DATE '2007-04-10',
          DATE '2007-06-10', 75, 7654);

INSERT INTO sales.odetail
  VALUES (124, 4130, 25000, 2);

UPDATE invent.partloc
  SET qty_on_hand = qty_on_hand - 2
  WHERE partnum = 4130 AND loc_code = 'K43';

ROLLBACK WORK;
```

ROLLBACK WORK cancels the insert and update that occurred during the transaction.

SELECT Statement

- "Syntax Description of SELECT"
- "Considerations for SELECT"
- "Considerations for Select List"
- "Considerations for GROUP BY"
- "Considerations for ORDER BY"
- "Considerations for UNION"
- "Examples of SELECT"

The SELECT statement is a DML statement that retrieves values from tables, views, and derived tables determined by the evaluation of query expressions, or joined tables.

```
sql-query is:
      query-specification
    | query-expr-and-order

query-specification is:
SELECT [ "[" ANY N "]" | "[" FIRST N "]" ] [ALL | DISTINCT] select-list
   FROM table-ref [,table-ref]...
   [WHERE search-condition]
   [SAMPLE sampling-method]
   [TRANSPOSE transpose-set [transpose-set]...
      [KEY BY key-colname]]...
   [SEQUENCE BY colname [ASC[ENDING] | DESC[ENDING]]
      [,colname [ASC[ENDING] | DESC[ENDING]]]...]
   [GROUP BY {colname | colnum} [,{colname | colnum}]...]
   [HAVING search-condition]
   [access-clause ]
   [mode-clause]

query-expr-and-order is:
   query-expr [order-by-clause] [access-clause] [mode-clause]

query-expr is:
     query-primary
   | query-expr UNION [ALL] query-primary

query-primary is:
   simple-table | (query-expr)

simple-table is:
     VALUES (row-value-const) [,(row-value-const)]...
   | TABLE table
   | query-specification

row-value-const is:
     row-subquery
   | {expression | NULL} [,{expression | NULL}]...
order-by-clause is:
   [ORDER BY {colname | colnum} [ASC[ENDING] | DESC[ENDING]]
      [,{colname | colnum} [ASC[ENDING] | DESC[ENDING]]]...]
   [access-clause]

access clause is:
   [FOR] access-option ACCESS
```

```
access-option is:
     READ COMMITTED

[LIMIT num]

select-list is:
   * | select-sublist [,select-sublist]...

select-sublist is:
     corr.* | [corr.]single-col [[AS]name] | col-expr [[AS] name]

table-ref is:
     table [[AS] corr [(col-expr-list)]]
   | view  [[AS] corr [(col-expr-list)]]
   | (query-expr) [AS] corr [(col-expr-list)]
   | (delete-statement [RETURN select-list])
        [AS] corr [(col-expr-list)]
   | (update-statement [RETURN select-list])
        [AS] corr [(col-expr-list)]
   | (insert-statement) [AS] corr [(col-expr-list)]
   | joined-table

joined-table is:
     table-ref [join-type] JOIN table-ref join-spec
   | table-ref NATURAL [join-type] JOIN table-ref
   | table-ref CROSS JOIN table-ref
   | (joined-table)

join-type is:
   INNER | LEFT [OUTER] | RIGHT [OUTER] | FULL [OUTER]

join-spec is:
   ON search-condition

sampling-method is:
     RANDOM percent-size
   | FIRST rows-size
           [SORT BY colname [ASC[ENDING] | DESC[ENDING]]
             [,colname [ASC[ENDING] | DESC[ENDING]]...]
   | PERIODIC rows-size EVERY number-rows ROWS
           [SORT BY colname [ASC[ENDING] | DESC[ENDING]]
             [,colname [ASC[ENDING] | DESC[ENDING]]...]

percent-size is:
     percent-result PERCENT [ROWS]
    | BALANCE WHEN condition
              THEN percent-result PERCENT [ROWS]
             [WHEN condition THEN percent-result PERCENT [ROWS]]...
             [ELSE percent-result PERCENT [ROWS]] END

rows-size is:
     number-rows ROWS
   | BALANCE WHEN condition THEN number-rows ROWS
             [WHEN condition THEN number-rows ROWS]...
             [ELSE number-rows ROWS] END

transpose-set is:
   transpose-item-list AS transpose-col-list

transpose-item-list is:
   expression-list | (expression-list) [,(expression-list)]...
```

```
expression-list is:
   expression [,expression]...

transpose-col-list is:
   colname | (colname-list)

colname-list is:
   colname [,colname]...
```

Syntax Description of SELECT

"[" ANY *N* "]" | "[" FIRST *N* "]"

specifies that *N* rows are to be returned (assuming the table has at least *N* rows and that the qualification criteria specified in the WHERE clause, if any, would select at least *N* rows) and you do not care which *N* rows are chosen (out of the qualified rows) to actually be returned.

You must enclose ANY N or FIRST N in square brackets ([]). The quotation marks (" ") around each square bracket in the syntax diagram indicate that the bracket is a required character that you must type as shown (for example, [ANY 10] or [FIRST 5]). Do not include quotation marks in ANY or FIRST clauses.

[FIRST *N*] is different from [ANY *N*] only if you use ORDER BY on any of the columns in the select list to sort the result table of the SELECT statement. *N* is an unsigned numeric literal with no scale. If *N* is greater than the number of rows in the table, all rows are returned. [ANY *N*] and [FIRST *N*] are disallowed in nested SELECT statements and on either side of a UNION operation.

ALL | DISTINCT

specifies whether to retrieve all rows whose columns are specified by the *select-list* (ALL) or only rows that are not duplicates (DISTINCT). Nulls are considered equal for the purpose of removing duplicates. The default is ALL.

select-list

specifies the columns or column expressions to select from the table references in the FROM clause.

*

specifies all columns in a table, view, joined table, or derived table determined by the evaluation of a query expression, as specified in the FROM clause.

corr.*

specifies all columns of specific table references by using the correlation name *corr* of the table references, as specified in the FROM clause. See "Correlation Names" (page 196).

corr.single-col [[AS] *name*]

specifies one column of specific table references by using the correlation name of the table reference, as specified in the FROM clause. See "Correlation Names" (page 196). By using the AS clause, you can associate the column with a *name*. *name* is an SQL identifier. See "Identifiers" (page 221).

single-col [[AS] *name*]

specifies a column. By using the AS clause, you can associate the column with a *name*. *name* is an SQL identifier. See "Identifiers" (page 221).

col-expr [[AS] *name*]

specifies a derived column determined by the evaluation of an SQL value expression in the list. By using the AS clause, you can associate a derived column, *col-expr*, with a *name*. *name* is an SQL identifier. See "Identifiers" (page 221).

See the discussion of limitations in "Considerations for Select List" (page 148).

FROM *table-ref* [,*table-ref*]...

specifies a list of tables, views, derived tables, or joined tables that determine the contents of an intermediate result table from which Trafodion SQL returns the columns you specify in *select-list*.

If you specify only one *table-ref*, the intermediate result table consists of rows derived from that table reference. If you specify more than one *table-ref*, the intermediate result table is the cross-product of result tables derived from the individual table references.

table [[AS] *corr* [(*col-expr-list*)]] | *view* [[AS] *corr* [(*col-expr-list*)]] | (*query-expr*) [AS] *corr* [(*col-expr-list*)] | (*delete-statement* [RETURN *select-list*]) [AS] *corr* [(*col-expr-list*)] | (*update-statement* [RETURN *select-list*]) [AS] *corr* [(*col-expr-list*)] | (*insert-statement*) [AS] *corr* [(*col-expr-list*)] | *joined-table*

specifies a *table-ref* as a single table, view, derived table determined by the evaluation of a query expression, or a joined table.

You can specify this optional clause for a table or view. This clause is required for a derived table:

[AS] *corr* [(*col-expr-list*)]

specifies a correlation name, *corr*, for the preceding table reference *table-ref* in the FROM clause. See "Correlation Names" (page 196).

col-expr [[AS] *name*] [,*col-expr* [[AS] *name*]]...

specifies the items in *col-expr-list*, a list of derived columns. By using the AS clause, you can associate a derived column, *col-expr*, with a *name*. *name* is an SQL identifier. See "Identifiers" (page 221).

For the specification of a query expression, see the syntax diagram for *query-expr* above.

(*delete-statement* [RETURN *select-list*]) [AS] *corr* [(*col-expr-list*)]

enables an application to read and delete rows with a single operation. For the syntax of *delete-statement*, see the "DELETE Statement" (page 86).

RETURN *select-list*

specifies the columns or column expressions returned from the deleted row. The items in the *select-list* can be of these forms:

[OLD.]*

specifies the row from the old table exposed by the embedded delete. The old table refers to column values before the delete operation. NEW is not allowed.

An implicit `OLD.*` return list is assumed for a delete operation that does not specify a return list.

col-expr [[AS] *name*]

specifies a derived column determined by the evaluation of an SQL value expression in the list. Any column referred to in a value expression is from the row in the old table exposed by the delete. The old table refers to column values before the delete operation.

By using the AS clause, you can associate a derived column, *col-expr*, with a *name*. *name* is an SQL identifier. See "Identifiers" (page 221).

[AS] *corr* [(*col-expr-list*)]

specifies a correlation name, *corr*, and an optional column list for the preceding items in the select list `RETURN` *select-list*. See "Correlation Names" (page 196).

(*update-statement* [RETURN *select-list*]) [AS] *corr* [(*col-expr-list*)]

enables an application to read and update rows with a single operation. For the syntax of *update-statement*, see the "UPDATE Statement" (page 169).

RETURN *select-list*

specifies the columns or column expressions returned from the updated row. The items in the *select-list* can be of these forms:

[OLD.| NEW.]*

specifies the row from the old or new table exposed by the update. The old table refers to column values before the update operation; the new table refers to column values after the update operation. If a column has not been updated, the new value is equivalent to the old value.

An implicit `NEW.*` return list is assumed for an update operation that does not specify a return list.

col-expr [[AS] *name*]

specifies a derived column determined by the evaluation of an SQL value expression in the list. Any column referred to in a value expression can be specified as being from the row in the old table exposed by the update or can be specified as being from the row in the new table exposed by the update.

For example: `RETURN old.empno,old.salary,new.salary, (new.salary - old.salary).`

By using the AS clause, you can associate a derived column, *col-expr*, with a *name*. *name* is an SQL identifier. See "Identifiers" (page 221).

[AS] *corr* [(*col-expr-list*)]

specifies a correlation name, *corr*, and an optional column list for the preceding items in the select list `RETURN` *select-list*. See "Correlation Names" (page 196).

For example:

```
RETURN old.empno,old.salary,new.salary,
   (new.salary - old.salary)
AS emp (empno, oldsalary, newsalary, increase).
```

(*insert-statement*) [AS] *corr* [(*col-expr-list*)]

For the syntax of *insert-statement*, see the "INSERT Statement" (page 118).

[AS] *corr* [(*col-expr-list*)]

specifies a correlation name, *corr*, and an optional column list. See "Correlation Names" (page 196).

joined-table

A *joined-table* can be specified as:

```
table-ref [join-type] JOIN table-ref join-spec
| table-ref NATURAL [join-type] JOIN table-ref
| table-ref CROSS JOIN table-ref
| (joined-table)
```

join-type `is: INNER | LEFT [OUTER] | RIGHT [OUTER] | FULL [OUTER]`

is a joined table. You specify the *join-type* by using the CROSS, INNER, OUTER, LEFT, RIGHT, and FULL keywords. If you omit the optional OUTER keyword and use LEFT, RIGHT, or FULL in a join, Trafodion SQL assumes the join is an outer join.
If you specify a CROSS join as the *join-type*, you cannot specify a NATURAL join or a *join-spec*.
If you specify an INNER, LEFT, RIGHT, or FULL join as the *join-type* and you do not specify a NATURAL join, you must use an ON clause as the *join-spec*, as follows:
Subqueries are not allowed in the join predicate of FULL OUTER JOIN.

ON *search-condition*

specifies a *search-condition* for the join. Each column reference in *search-condition* must be a column that exists in either of the two result tables derived from the table references to the left and right of the JOIN keyword. A join of two rows in the result tables occurs if the condition is satisfied for those rows.

The type of join and the join specification if used determine which rows are joined from the two table references, as follows:

table-ref CROSS JOIN *table-ref*

joins each row of the left *table-ref* with each row of the right *table-ref*.

table-ref NATURAL JOIN *table-ref*

joins rows only where the values of all columns that have the same name in both tables match. This option is equivalent to NATURAL INNER.

table-ref NATURAL LEFT JOIN *table-ref*

joins rows where the values of all columns that have the same name in both tables match, plus rows from the left *table-ref* that do not meet this condition.

table-ref NATURAL RIGHT JOIN *table-ref*

joins rows where the values of all columns that have the same name in both tables match, plus rows from the right *table-ref* that do not meet this condition.

table-ref NATURAL FULL JOIN *table-ref*

joins rows where the values of all columns that have the same name in both tables match, plus rows from either side that do not meet this condition, filling in NULLs for missing values.

table-ref JOIN *table-ref join-spec*

joins only rows that satisfy the condition in the *join-spec* clause. This option is equivalent to INNER JOIN ... ON.

table-ref LEFT JOIN *table-ref join-spec*

joins rows that satisfy the condition in the *join-spec* clause, plus rows from the left *table-ref* that do not satisfy the condition.

table-ref RIGHT JOIN *table-ref join-spec*

joins rows that satisfy the condition in the *join-spec* clause, plus rows from the right *table-ref* that do not satisfy the condition.

table-ref FULL OUTER JOIN *table-ref join-spec*

combines the results of both left and right outer joins. These joins show records from both tables and fill in NULLs for missing matches on either side

simple-table

A *simple-table* can be specified as:

```
  VALUES (row-value-const) [,(row-value-const)]...
| TABLE table
| query-specification
```

A *simple-table* can be a table value constructor. It starts with the VALUES keyword followed by a sequence of row value constructors, each of which is enclosed in parentheses. A *row-value-const* is a list of expressions (or NULL) or a row subquery (a subquery that returns a single row of column values). An operand of an expression cannot reference a column (except when the operand is a scalar subquery returning a single column value in its result table).

The use of NULL as a *row-value-const* element is a Trafodion SQL extension.

A *simple-table* can be specified by using the TABLE keyword followed by a table name, which is equivalent to the query specification `SELECT * FROM table`.

A *simple-table* can be a *query-specification*—that is, a SELECT statement consisting of SELECT ... FROM ... with optionally the WHERE, SAMPLE, TRANSPOSE, SEQUENCE BY, GROUP BY, and HAVING clauses.

WHERE *search-condition*

specifies a *search-condition* for selecting rows. See "Search Condition" (page 250). The WHERE clause cannot contain an aggregate (set) function.

The *search-condition* is applied to each row of the result table derived from the table reference in the FROM clause or, in the case of multiple table references, the cross-product of result tables derived from the individual table references.

Each column you specify in *search-condition* is typically a column in this intermediate result table. In the case of nested subqueries used to provide comparison values, the column can also be an outer reference. See"Subquery" (page 252).

To comply with ANSI standards, Trafodion SQL does not move aggregate predicates from the WHERE clause to a HAVING clause and does not move non-aggregate predicates from the HAVING clause to the WHERE clause.

SAMPLE *sampling-method*

specifies the sampling method used to select a subset of the intermediate result table of a SELECT statement. Each of the methods uses a sampling size. The three sampling methods—random, first, and periodic—are specified as:

RANDOM *percent-size*

directs Trafodion SQL to choose rows randomly (each row having an unbiased probability of being chosen) without replacement from the result table. The sampling size is determined by using a percent of the result table.

FIRST *rows-size* [SORT BY *colname* [,*colname*]...]

directs Trafodion SQL to choose the first *rows-size* rows from the sorted result table. The sampling size is determined by using the specified number of rows.

PERIODIC *rows-size* EVERY *number-rows* ROWS [SORT BY *colname* [,*colname*]...]

directs Trafodion SQL to choose the first rows from each block (period) of contiguous sorted rows. The sampling size is determined by using the specified number of rows chosen from each block.

SAMPLE is a Trafodion SQL extension. See "SAMPLE Clause" (page 261).

TRANSPOSE *transpose-set*[*transpose-set*]... [KEY BY *key-colname*]

specifies the *transpose-sets* and an optional key clause within a TRANSPOSE clause. You can use multiple TRANSPOSE clauses in a SELECT statement.

transpose-item-list AS *transpose-col-list*

specifies a *transpose-set*. You can use multiple transpose sets within a TRANSPOSE clause. The TRANSPOSE clause generates, for each row of the source table derived from the table reference or references in the FROM clause, a row for each item in each *transpose-item-list* of all the transpose sets.

The result table of a TRANSPOSE clause has all the columns of the source table plus a value column or columns, as specified in each *transpose-col-list* of all the transpose sets, and an optional key column *key-colname*.

KEY BY *key-colname*

optionally specifies an optional key column *key-colname*. It identifies which expression the value in the transpose column list corresponds to by its position in the *transpose-item-list*. *key-colname* is an SQL identifier. The data type is exact numeric, and the value is NOT NULL.

TRANSPOSE is a Trafodion SQL extension. See "TRANSPOSE Clause" (page 271).

SEQUENCE BY *colname* [ASC[ENDING] | DESC[ENDING]] [,*colname* [ASC[ENDING] | DESC[ENDING]]]...

specifies the order in which to sort the rows of the intermediate result table for calculating sequence functions. You must include a SEQUENCE BY clause if you include a sequence function in *select-list*. Otherwise, Trafodion SQL returns an error. Further, you cannot include a SEQUENCE BY clause if no sequence function is in *select-list*.

colname

names a column in *select-list* or a column in a table reference in the FROM clause of the SELECT statement. *colname* is optionally qualified by a table, view, or correlation name; for example, CUSTOMER.CITY.

ASC | DESC

specifies the sort order. The default is ASC. When Trafodion SQL orders an intermediate result table on a column that can contain null, nulls are considered equal to one another but greater than all other nonnull values.

GROUP BY [*col-expr*] {*colname* | *colnum*} [,{*colname* | *colnum*}]...]

specifies grouping columns that define a set of groups for the result table of the SELECT statement. The expression in the GROUP BY clause must be exactly the same as the expression in the select list. These columns must appear in the list of columns in the table references in the FROM clause of the SELECT statement.

If you include a GROUP BY clause, the columns you refer to in the *select-list* must be grouping columns or arguments of an aggregate (or set) function.

The grouping columns define a set of groups in which each group consists of rows with identical values in the specified columns. The column names can be qualified by a table or view name or a correlation name; for example, CUSTOMER.CITY.

For example, if you specify AGE, the result table contains one group of rows with AGE equal to 40 and one group of rows with AGE equal to 50. If you specify AGE and then JOB, the result table contains one group for each age and, within each age group, subgroups for each job code.

You can specify GROUP BY using ordinals to refer to the relative position within the SELECT list. For example, `GROUP BY 3, 2, 1`.

For grouping purposes, all nulls are considered equal to one another. The result table of a GROUP BY clause can have only one null group.

See "Considerations for GROUP BY" (page 148).

HAVING *search-condition*

specifies a *search-condition* to apply to each group of the grouped table resulting from the preceding GROUP BY clause in the SELECT statement.

To comply with ANSI standards, Trafodion SQL does not move aggregate predicates from the WHERE clause to a HAVING clause and does not move non-aggregate predicates from the HAVING clause to the WHERE clause.

If no GROUP BY clause exists, the *search-condition* is applied to the entire table (which consists of one group) resulting from the WHERE clause (or the FROM clause if no WHERE clause exists).

In *search-condition*, you can specify any column as the argument of an aggregate (or set) function; for example, AVG (SALARY). An aggregate function is applied to each group in the grouped table.

A column that is not an argument of an aggregate function must be a grouping column. When you refer to a grouping column, you are referring to a single value because each row in the group contains the same value in the grouping column.

See "Search Condition" (page 250).

[FOR] *access-option* ACCESS

specifies the *access-option* when accessing data specified by the SELECT statement or by a table reference in the FROM clause derived from the evaluation of a query expression that is a SELECT statement. See "Data Consistency and Access Options" (page 25).

READ COMMITTED

specifies that any data accessed must be from committed rows.

UNION [ALL] *select-stmt*

specifies a set union operation between the result table of a SELECT statement and the result table of another SELECT statement.

The result of the union operation is a table that consists of rows belonging to either of the two contributing tables. If you specify UNION ALL, the table contains all the rows retrieved by each SELECT statement. Otherwise, duplicate rows are removed.

The select lists in the two SELECT statements of a union operation must have the same number of columns, and columns in corresponding positions within the lists must have compatible data types. The select lists must not be preceded by `[ANY N]` or `[FIRST N]`.

The number of columns in the result table of the union operation is the same as the number of columns in each select list. The column names in the result table of the union are the same as the corresponding names in the select list of the left SELECT statement. A column resulting from the union of expressions or constants has the name (EXPR).

See "Considerations for UNION" (page 149).

ORDER BY {*colname* | *colnum*} [ASC[ENDING] | DESC[ENDING]] [,{*colname* | *colnum*} [ASC[ENDING] | DESC[ENDING]]]...

specifies the order in which to sort the rows of the final result table.

colname

names a column in *select-list* or a column in a table reference in the FROM clause of the SELECT statement. *colname* is optionally qualified by a table, view, or correlation name; for example, CUSTOMER.CITY. If a column has been aliased to another name you must use the alias name.

colnum

specifies a column by its position in *select-list*. Use *colnum* to refer to unnamed columns, such as derived columns.

ASC | DESC

specifies the sort order. The default is ASC. For ordering a result table on a column that can contain null, nulls are considered equal to one another but greater than all other nonnull values.

See "Considerations for ORDER BY" (page 148).

LIMIT *num*

limits the number of rows returned by the query with no limit applied if *num* is null or less than zero. The LIMIT clause is executed after the ORDER BY clause to support TopN queries.

Considerations for SELECT

Authorization Requirements

SELECT requires authority to read all views and tables referred to in the statement, including the underlying tables of views referred to in the statement.

Use of Views With SELECT

When a view is referenced in a SELECT statement, the specification that defines the view is combined with the statement. The combination can cause the SELECT statement to be invalid. If you receive

an error message that indicates a problem but the SELECT statement seems to be valid, check the view definition.

For example, suppose that the view named AVESAL includes column A defined as AVG (X). The SELECT statement that contains MAX (A) in its select list is invalid because the select list actually contains MAX (AVG (X)), and an aggregate function cannot have an argument that includes another aggregate function.

Join Limits

NOTE: We recommend that you limit the number of tables in a join to a maximum of 64, which includes base tables of views referenced in joins. Queries with joins that involve a larger number of tables are not guaranteed to compile.

Object Names in SELECT

You can use fully qualified names only in the FROM clause of a SELECT statement.

AS and ORDER BY Conflicts

When you use the AS verb to rename a column in a SELECT statement, and the ORDER BY clause uses the original column name, the query fails. If a column has been aliased to another name, you must use the alias name. The ANSI standard does not support this type of query.

Restrictions on Embedded Inserts

- An embedded INSERT cannot be used in a join.
- An embedded INSERT cannot appear in a subquery.
- An embedded INSERT statement cannot have a subquery in the WHERE clause.
- An INSERT statement cannot contain an embedded INSERT statement.
- A union between embedded INSERT expressions is not supported.
- Declaring a cursor on an embedded INSERT statement is not supported.

DISTINCT Aggregate Functions

An aggregate function can accept an argument specified as DISTINCT, which eliminates duplicate values before the aggregate function is applied. For a given grouping, multiple DISTINCT aggregates are allowed and can be used with non distinct aggregates. A restriction exists that DISTINCT STDDEV and VARIANCE cannot be used with multiple DISTINCT aggregates.

Limitations of DISTINCT Aggregates

- No limit exists to the number of distinct aggregates.
- Distinct STDDEV and distinct VARIANCE are not supported with multiple distinct aggregates. For example, this statement will result in an error.

```
SELECT sum(distinct a), stddev(distinct b) from T group by d;
```

Examples of Multiple Distinct Aggregates

- This statement contains distinct aggregates:

```
SELECT sum(distinct a), count(distinct b), avg(distinct c)
  from T group by d;
```

- This statement does not contain multiple distincts. Because each distinct aggregate is on the same column (a), this is treated as one distinct value.

```
SELECT sum(distinct a), count(distinct a), avg(distinct a)
  from T group by d;
```

- This statement shows that multiple distinct aggregates can be used with non distinct aggregates:

```
SELECT sum(distinct a), avg(distinct b), sum(c)
  from T group by d;
```

Considerations for Select List

- The * and *corr*.* forms of a *select-list* specification are convenient. However, such specifications make the order of columns in the SELECT result table dependent on the order of columns in the current definition of the referenced tables or views.
- A *col-expr* is a single column name or a derived column. A derived column is an SQL value expression; its operands can be numeric, string, datetime, or interval literals, columns, functions (including aggregate functions) defined on columns, scalar subqueries, CASE expressions, or CAST expressions. Any single columns named in *col-expr* must be from tables or views specified in the FROM clause. For a list of aggregate functions, see "Aggregate (Set) Functions" (page 278).
- If *col-expr* is a single column name, that column of the SELECT result table is a named column. All other columns are unnamed columns in the result table (and have the (EXPR) heading) unless you use the AS clause to specify a name for a derived column.

Considerations for GROUP BY

- If you include a GROUP BY clause, the columns you refer to in the *select-list* must be either grouping columns or arguments of an aggregate (or set) function. For example, if AGE is not a grouping column, you can refer to AGE only as the argument of a function, such as AVG (AGE).
- The expression in the GROUP BY clause must be exactly the same as the expression in the select list. An error will be returned if it is not. It cannot contain aggregate functions or subqueries.
- If the value of *col-expr* is a numeric constant, it refers to the position of the select list item and is treated as the current GROUP BY using the ordinal feature.
- You can specify GROUP BY using ordinals to refer to the relative position within the SELECT list. For example, `GROUP BY 3, 2, 1`.
- If you do not include a GROUP BY clause but you specify an aggregate function in the *select-list*, all rows of the result table form the one and only group. The result of AVG, for example, is a single value for the entire table.

Considerations for ORDER BY

When you specify an ORDER BY clause and its ordering columns, consider:

- ORDER BY is allowed only in the outer level of a query or in the SELECT part of an INSERT/SELECT statement. It is not allowed inside nested SELECT expressions, such as subqueries.
- If you specify DISTINCT, the ordering column must be in *select-list*.
- If you specify a GROUP BY clause, the ordering column must also be a grouping column.

- If an ORDER BY clause applies to a union of SELECT statements, the ordering column must be explicitly referenced, and not within an aggregate function or an expression, in the *select-list* of the leftmost SELECT statement.
- SQL does not guarantee a specific or consistent order of rows unless you specify an ORDER BY clause. ORDER BY can reduce performance, however, so use it only if you require a specific order.

Considerations for UNION

Suppose that the contributing SELECT statements are named SELECT1 and SELECT2, the contributing tables resulting from the SELECT statements are named TABLE1 and TABLE2, and the table resulting from the UNION operation is named RESULT.

Characteristics of the UNION Columns

For columns in TABLE1 and TABLE2 that contribute to the RESULT table:

- If both columns contain character strings, the corresponding column in RESULT contains a character string whose length is equal to the greater of the two contributing columns.
- If both columns contain variable-length character strings, RESULT contains a variable-length character string whose length is equal to the greater of the two contributing columns.
- If both columns are of exact numeric data types, RESULT contains an exact numeric value whose precision and scale are equal to the greater of the two contributing columns.
- If both columns are of approximate numeric data types, RESULT contains an approximate numeric value whose precision is equal to the greater of the two contributing columns.
- If both columns are of datetime data type (DATE, TIME, or TIMESTAMP), the corresponding column in RESULT has the same data type.
- If both columns are INTERVAL data type and both columns are year-month or day-time, RESULT contains an INTERVAL value whose range of fields is the most significant start field to the least significant end field of the INTERVAL fields in the contributing columns. (The year-month fields are YEAR and MONTH. The day-time fields are DAY, HOUR, MINUTE, and SECOND.)

  For example, suppose that the column in TABLE1 has the data type INTERVAL HOUR TO MINUTE, and the column in TABLE2 has the data type INTERVAL DAY TO HOUR. The data type of the column resulting from the union operation is INTERVAL DAY TO MINUTE.
- If both columns are described with NOT NULL, the corresponding column of RESULT cannot be null. Otherwise, the column can be null.

ORDER BY Clause and the UNION Operator

In a query containing a UNION operator, the ORDER BY clause defines an ordering on the result of the union. In this case, the SELECT statement cannot have an individual ORDER BY clause.

You can specify an ORDER BY clause only as the last clause following the final SELECT statement (SELECT2 in this example). The ORDER BY clause in RESULT specifies the ordinal position of the sort column either by using an integer or by using the column name from the select list of SELECT1.

This SELECT statement shows correct use of the ORDER BY clause:

```
SELECT A FROM T1 UNION SELECT B FROM T2 ORDER BY A
```

This SELECT statement is incorrect because the ORDER BY clause does not follow the final SELECT statement:

```
SELECT A FROM T1 ORDER BY A UNION SELECT B FROM T2
```

This SELECT statement is also incorrect:

```
SELECT A FROM T1 UNION (SELECT B FROM T2 ORDER BY A)
```

Because the subquery (SELECT B FROM T2...) is processed first, the ORDER BY clause does not follow the final SELECT.

GROUP BY Clause, HAVING Clause, and the UNION Operator

In a query containing a UNION operator, the GROUP BY or HAVING clause is associated with the SELECT statement it is a part of (unlike the ORDER BY clause, which can be associated with the result of a union operation). The groups are visible in the result table of the particular SELECT statement. The GROUP BY and HAVING clauses cannot be used to form groups in the result of a union operation.

UNION ALL and Associativity

The UNION ALL operation is left associative, meaning that these two queries return the same result:

```
(SELECT * FROM TABLE1 UNION ALL
  SELECT * FROM TABLE2) UNION ALL SELECT * FROM TABLE3;
SELECT * FROM TABLE1 UNION ALL
   (SELECT * FROM TABLE2 UNION ALL SELECT * FROM TABLE3);
```

If both the UNION ALL and UNION operators are present in the query, the order of evaluation is always from left to right. A parenthesized union of SELECT statements is evaluated first, from left to right, followed by the remaining union of SELECT statements.

Examples of SELECT

- Retrieve information from the EMPLOYEE table for employees with a job code greater than 500 and who are in departments with numbers less than or equal to 3000, displaying the results in ascending order by job code:

```
SELECT jobcode, deptnum, first_name, last_name, salary
FROM persnl.employee
WHERE jobcode > 500 AND deptnum <= 3000
ORDER BY jobcode;

JOBCODE  DEPTNUM  FIRST_NAME       LAST_NAME        SALARY
-------  -------  ---------------  -----------  ----------
    600  1500     JONATHAN         MITCHELL       32000.00
    600  1500     JIMMY            SCHNEIDER      26000.00
    900  2500     MIRIAM           KING           18000.00
    900  1000     SUE              CRAMER         19000.00
  . . .
```

- Display selected rows grouped by job code in ascending order:

```
SELECT jobcode, AVG(salary)
FROM persnl.employee
WHERE jobcode > 500 AND deptnum <= 3000
GROUP BY jobcode
ORDER BY jobcode;

JOBCODE  EXPR
-------  ----------------------
    600                29000.00
    900                25100.00

--- 2 row(s) selected.
```

This select list contains only grouping columns and aggregate functions. Each row of the output summarizes the selected data within one group.

- Select data from more than one table by specifying the table names in the FROM clause and specifying the condition for selecting rows of the result in the WHERE clause:

```
SELECT jobdesc, first_name, last_name, salary
```

```
FROM persnl.employee E, persnl.job J
WHERE E.jobcode = J.jobcode AND
  E.jobcode IN (900, 300, 420);

JOBDESC        FIRST_NAME    LAST_NAME         SALARY
-------------  ------------  ---------------  -----------
SALESREP       TIM           WALKER              32000.00
SALESREP       HERBERT       KARAJAN             29000.00
...
ENGINEER       MARK          FOLEY               33000.00
ENGINEER       MARIA         JOSEF               18000.10
...
SECRETARY      BILL          WINN                32000.00
SECRETARY      DINAH         CLARK               37000.00
...

--- 27 row(s) selected.
```

This type of condition is sometimes called a join predicate. The query first joins the EMPLOYEE and JOB tables by combining each row of the EMPLOYEE table with each row of the JOB table; the intermediate result is the Cartesian product of the two tables.

This join predicate specifies that any row (in the intermediate result) with equal job codes is included in the result table. The WHERE condition further specifies that the job code must be 900, 300, or 420. All other rows are eliminated.

The four logical steps that determine the intermediate and final results of the previous query are:

1. Join the tables.

EMPLOYEE Table			**JOB Table**	
EMPNUM ...	JOBCODE ...	SALARY	JOBCODE	JOBDESC

2. Drop rows with unequal job codes.

EMPLOYEE Table			**JOB Table**	
EMPNUM ...	JOBCODE ...	SALARY	JOBCODE	JOBDESC
1	100	175500	100	MANAGER
. . .	. . .	. . .	. . .	. . .
75	300	32000	300	SALESREP
. . .	. . .	. . .	. . .	. . .
178	900	28000	900	SECRETARY
. . .	. . .	. . .	. . .	. . .
207	420	33000	420	ENGINEER
. . .	. . .	. . .	. . .	. . .
568	300	39500	300	SALESREP

3. Drop rows with job codes not equal to 900, 300, or 420.

EMPLOYEE Table			**JOB Table**	
EMPNUM ...	JOBCODE ...	SALARY	JOBCODE	JOBDESC
75	300	32000	300	SALESREP

EMPLOYEE Table			JOB Table	
. . .	. . .	. . .	. . .	. . .
178	900	28000	900	SECRETARY
. . .	. . .	. . .	. . .	. . .
207	420	33000	420	ENGINEER
. . .	. . .	. . .	. . .	. . .
568	300	39500	300	SALESREP

4. Process the select list, leaving only four columns.

JOBDESC	FIRST_NAME	LAST_NAME	SALARY
SALESREP	TIM	WALKER	32000
. . .	. . .	. . .	. . .
SECRETARY	JOHN	CHOU	28000
. . .	. . .	. . .	. . .
ENGINEER	MARK	FOLEY	33000
. . .	. . .	. . .	. . .
SALESREP	JESSICA	CRINER	39500

The final result is shown in the output:

```
JOBDESC        FIRST_NAME    LAST_NAME         SALARY
-------------  ------------  ---------------   -----------
SALESREP       TIM           WALKER            32000.00
...
SECRETARY      JOHN          CHOU              28000.00
...
```

- Select from three tables, group the rows by job code and (within job code) by department number, and order the groups by the maximum salary of each group:

```
SELECT E.jobcode, E.deptnum, MIN (salary), MAX (salary)
FROM persnl.employee E,
  persnl.dept D, persnl.job J
WHERE E.deptnum = D.deptnum AND E.jobcode = J.jobcode
  AND E.jobcode IN (900, 300, 420)
GROUP BY E.jobcode, E.deptnum
ORDER BY 4;

JOBCODE  DEPTNUM  (EXPR)       (EXPR)
-------  -------  -----------  -----------
    900     1500     17000.00     17000.00
    900     2500     18000.00     18000.00
    ...
    300     3000     19000.00     32000.00
    900     2000     32000.00     32000.00
    ...
    300     3200     22000.00     33000.10
    420     4000     18000.10     36000.00
    ...

--- 16 row(s) selected.
```

Only job codes 300, 420, and 900 are selected. The minimum and maximum salary for the same job in each department are computed, and the rows are ordered by maximum salary.

- Select from two tables that have been joined by using an INNER JOIN on matching part numbers:

```
SELECT OD.*, P.*
FROM sales.odetail OD INNER JOIN sales.parts P
ON OD.partnum = P.partnum;

Order/Num  Part/Num  Unit/Price  Qty/Ord  Part/Num  Part Description  PRICE  Qty/Avail
---------  --------  ----------  ---------  --------
-----------------  ----------  ----------
    400410       212       2450.00          12       212
PCSILVER, 20 MB          2500.00         3525

    500450       212       2500.00           8       212
PCSILVER, 20 MB          2500.00         3525

    100210       244       3500.00           3       244
PCGOLD, 30 MB            3000.00         4426

    800660       244       3000.00           6       244
PCGOLD, 30 MB            3000.00         4426
       ...       ...           ...         ...       ...
...                         ...          ...

--- 72 row(s) selected.
```

- Select from three tables and display them in employee number order. Two tables are joined by using a LEFT JOIN on matching department numbers, then an additional table is joined on matching jobcodes:

```
SELECT empnum, first_name, last_name, deptname, location, jobdesc
  FROM employee e LEFT JOIN dept d ON e.deptnum = d.deptnum
    LEFT JOIN job j ON e.jobcode = j.jobcode
  ORDER BY empnum;
```

- Suppose that the JOB_CORPORATE table has been created from the JOB table by using the CREATE LIKE statement. Form the union of these two tables:

```
SELECT * FROM job UNION SELECT * FROM job_corporate;

JOBCODE  JOBDESC
-------  ------------------
    100  MANAGER
    200  PRODUCTION SUPV
    250  ASSEMBLER
    300  SALESREP
    400  SYSTEM ANALYST
    420  ENGINEER
    450  PROGRAMMER
    500  ACCOUNTANT
    600  ADMINISTRATOR
    900  SECRETARY
    100  CORP MANAGER
    300  CORP SALESREP
    400  CORP SYSTEM ANALYS
    500  CORP ACCOUNTANT
    600  CORP ADMINISTRATOR
    900  CORP SECRETARY

--- 16 row(s) selected.
```

- A FULL OUTER JOIN combines the results of both left and right outer joins. These joins show records from both tables and fill in NULLs for missing matches on either side:

```
SELECT *
FROM employee
     FULL OUTER JOIN
     department
       ON employee.DepartmentID = department.DepartmentID;

LastName DepartmentID  DepartmentName  DepartmentID
-------  ------------  --------------  ------------
Smith              34  Clerical                34
Jones              33  Engineering             33
Robinson           34  Clerical                34
Jasper             36  NULL                  NULL
Steinberg          33  Engineering             33
Rafferty           31  Sales                   31
NULL             NULL  Marketing               35
```

- Present two ways to select the same data submitted by customers from California.

  The first way:

```
SELECT OD.ordernum, SUM (qty_ordered * price)
FROM sales.parts P, sales.odetail OD
WHERE OD.partnum = P.partnum AND OD.ordernum IN
  (SELECT O.ordernum
   FROM sales.orders O, sales.customer C
   WHERE O.custnum = C.custnum AND state = 'CALIFORNIA')
GROUP BY OD.ordernum;

ORDERNUM    (EXPR)
----------  ---------------------
    200490                1030.00
    300350               71025.00
    300380               28560.00
--- 3 row(s) selected.
```

  The second way:

```
SELECT OD.ordernum, SUM (qty_ordered * price)
FROM sales.parts P, sales.odetail OD
WHERE OD.partnum = P.partnum AND OD.ordernum IN
  (SELECT O.ordernum
   FROM sales.orders O
   WHERE custnum IN
    (SELECT custnum
     FROM sales.customer
     WHERE state = 'CALIFORNIA'))
GROUP BY OD.ordernum;

ORDERNUM    (EXPR)
----------  ---------------------
    200490                1030.00
    300350               71025.00
    300380               28560.00
--- 3 row(s) selected.
```

  The price for the total quantity ordered is computed for each order number.

- Show employees, their salaries, and the percentage of the total payroll that their salaries represent. Note the subquery as part of the expression in the select list:

```
SELECT empnum, first_name, last_name, salary,
CAST(salary * 100 / (SELECT SUM(salary) FROM persnl.employee)
  AS NUMERIC(4,2))
FROM persnl.employee
ORDER BY salary, empnum;
```

```
Employee/Number  First Name       Last Name            salary       (EXPR)
---------------  ---------------  -------------------  -----------  -------
            209  SUSAN            CHAPMAN                17000.00      .61
            235  MIRIAM           KING                   18000.00      .65
            224  MARIA            JOSEF                  18000.10      .65
...
             23  JERRY            HOWARD                137000.10     4.94
             32  THOMAS           RUDLOFF               138000.40     4.98
              1  ROGER            GREEN                 175500.00     6.33
...

--- 62 row(s) selected.
```

- Examples of using expressions in the GROUP BY clause:

```
SELECT a+1 FROM t GROUP BY a+1;
SELECT cast(a AS int) FROM t GROUP BY cast(a AS int);
SELECT a+1 FROM t GROUP BY 1;
```

- Examples of unsupported expressions in the GROUP BY clause:

```
SELECT sum(a) FROM t GROUP BY sum(a);
SELECT (SELECT a FROM t1) FROM t GROUP BY (SELECT a FROM t1);
SELECT a+1 FROM t GROUP BY 1+a;
```

SET SCHEMA Statement

- "Syntax Description of SET SCHEMA"
- "Considerations for SET SCHEMA"
- "Example of SET SCHEMA"

The SET SCHEMA statement sets the default logical schema for unqualified object names for the current SQL session.

```
SET SCHEMA default-schema-name
```

Syntax Description of SET SCHEMA

default-schema-name

specifies the name of a schema. See "Schemas" (page 249).

default-schema-name is an SQL identifier. For example, you can use MYSCHEMA or myschema or a delimited identifier "My_Schema". See "Identifiers" (page 221).

Considerations for SET SCHEMA

The default schema you specify with SET SCHEMA remains in effect until the end of the session or until you execute another SET SCHEMA statement. If you do not set a schema name for the session using SET SCHEMA, the default schema is SEABASE, which exists in the TRAFODION catalog.

For information on how to create a schema, see "Creating and Dropping Schemas" (page 249).

Example of SET SCHEMA

Set the default schema name:

```
SET SCHEMA myschema;
```

SET TRANSACTION Statement

- "Syntax Description of SET TRANSACTION"
- "Considerations for SET TRANSACTION"
- "Examples of SET TRANSACTION"

The SET TRANSACTION statement sets the autocommit attribute for transactions. It stays in effect until the end of the session or until the next SET TRANSACTION statement, whichever comes first. Therefore, the SET TRANSACTION statement can set the autocommit attribute of all subsequent transactions in the session.

```
SET TRANSACTION autocommit-option

autocommit-option is:
   AUTOCOMMIT [ON] | AUTOCOMMIT OFF
```

Syntax Description of SET TRANSACTION

autocommit-option

specifies whether Trafodion SQL commits or rolls back automatically at the end of statement execution. This option applies to any statement for which the system initiates a transaction.

If this option is set to ON, Trafodion SQL automatically commits any changes or rolls back any changes made to the database at the end of statement execution. AUTOCOMMIT is on by default at the start of a session.

If this option is set to OFF, the current transaction remains active until the end of the session unless you explicitly commit or rollback the transaction. AUTOCOMMIT is a Trafodion SQL extension; you cannot use in it with any other option.

Using the AUTOCOMMIT option in a SET TRANSACTION statement does not reset other transaction attributes that may have been specified in a previous SET TRANSACTION statement. Similarly, a SET TRANSACTION statement that does not specify the AUTOCOMMIT attribute does not reset this attribute.

Considerations for SET TRANSACTION

Implicit Transactions

Most DML statements are transaction initiating—the system automatically initiates a transaction when the statement begins executing.

The exceptions (statements that are not transaction initiating) are:

- COMMIT, FETCH, ROLLBACK, and SET TRANSACTION
- EXECUTE, which is transaction initiating only if the associated statement is transaction-initiating

Explicit Transactions

You can issue an explicit BEGIN WORK even if the autocommit option is on. The autocommit option is temporarily disabled until you explicitly issue COMMIT or ROLLBACK.

Examples of SET TRANSACTION

The following SET TRANSACTION statement turns off autocommit so that the current transaction remains active until the end of the session unless you explicitly commit or rollback the transaction. Trafodion SQL does not automatically commit or roll back any changes made to the database at the end of statement execution. Instead, Trafodion SQL commits all the changes when you issue the COMMIT WORK statement.

```
SET TRANSACTION AUTOCOMMIT OFF;
--- SQL operation complete.

BEGIN WORK;
--- SQL operation complete.

DELETE FROM persnl.employee
  WHERE empnum = 23;
--- 1 row(s) deleted.

INSERT INTO persnl.employee
  (empnum, first_name, last_name, deptnum, salary)
  VALUES (50, 'JERRY','HOWARD', 1000, 137000.00);
--- 1 row(s) inserted.

UPDATE persnl.dept
  SET manager = 50
  WHERE deptnum = 1000;
--- 1 row(s) updated.

COMMIT WORK;
--- SQL operation complete.
```

SHOWCONTROL Statement

The SHOWCONTROL statement displays the default attributes in effect.

SHOWCONTROL is a Trafodion SQL extension.

```
SHOWCONTROL {ALL | [QUERY] DEFAULT [attribute-name[, MATCH {FULL | PARTIAL }]]}
```

Syntax Description of SHOWCONTROL

ALL

displays all the hard-coded default attributes that have been set for the Trafodion instance.

[QUERY] DEFAULT

displays the CONTROL QUERY DEFAULT statements in effect for the session. For more information, see the"CONTROL QUERY DEFAULT Statement" (page 49)

attribute-name[, MATCH {FULL | PARTIAL }]

displays only the defaults that match, either fully or partially, the *attribute* used in CONTROL QUERY DEFAULT statements. The match is not case-sensitive. For descriptions of these attributes, see "Control Query Default (CQD) Attributes" (page 466).

MATCH FULL specifies that *attribute-name* must be the same as the attribute name used in a CONTROL QUERY DEFAULT statement. MATCH PARTIAL specifies that *attribute-name* must be included in the attribute name used in a CONTROL QUERY DEFAULT statement. The default is MATCH PARTIAL.

If *attribute-name* is a reserved word, such as MIN, MAX, or TIME, you must capitalize *attribute-name* and delimit it within double quotes ("). The only exceptions to this rule are the reserved words CATALOG and SCHEMA, which you can either capitalize and delimit within double quotes or specify without quotation marks.

Example of SHOWCONTROL

Issue multiple CONTROL QUERY DEFAULT statements followed by a SHOWCONTROL DEFAULT command:

```
CONTROL QUERY DEFAULT CACHE_HISTOGRAMS_REFRESH_INTERVAL '7200';

--- SQL operation complete.

CONTROL QUERY DEFAULT HIST_NO_STATS_REFRESH_INTERVAL '7200';

--- SQL operation complete.

SHOWCONTROL DEFAULT;

CONTROL QUERY DEFAULT
  CACHE_HISTOGRAMS_REFRESH_INTERVAL 7200
  HIST_NO_STATS_REFRESH_INTERVAL 7200

--- SQL operation complete.
```

SHOWDDL Statement

- "Syntax Description of SHOWDDL"
- "Considerations for SHOWDDL"
- "Examples of SHOWDDL"

The SHOWDDL statement describes the DDL syntax used to create an object as it exists in the metadata, or it returns a description of a user, role, or component in the form of a GRANT statement.

SHOWDDL is a Trafodion SQL extension.

```
SHOWDDL showddl-spec

showddl-spec
    [TABLE | LIBRARY | PROCEDURE] [schema-name.]object-name[, PRIVILEGES ]
   | COMPONENT component-name
   | USER database-username
   | ROLE role-name[, GRANTEES ]
```

Syntax Description of SHOWDDL

[*schema-name.*]*object-name*

specifies the ANSI name of an existing table, view, library, or procedure. See "Database Object Names" (page 198). If *object-name* is not fully qualified, SHOWDDL uses the default schema for the session.

PRIVILEGES

describes the PRIVILEGES associated with the object. If specified, privileges are displayed for an object in the form of GRANT statements.

component-name

specifies an existing component. Currently, the only valid component name is SQL_OPERATIONS.

database-username

specifies a registered database username.

role-name

specifies an existing role.

GRANTEES

displays all users who have been granted the role in the form of GRANT ROLE statements. This is an optional clause.

Considerations for SHOWDDL

- SHOWDDL can differ from the original DDL used to create an object.
- SHOWDDL can be used within TrafCI.
- SHOWDDL [TABLE | LIBRARY | PROCEDURE] displays the following information:
  - A constraint may be disabled.
  - A table may be offline.
  - An active DDL lock may exist on an object.
- SHOWDDL USER displays user information as a REGISTER USER statement.
- SHOWDDL ROLE displays the role information as a CREATE ROLE statement.

Required Privileges

To issue a SHOWDDL statement, one of the following must be true:

- You are DB__ROOT.
- You are the owner of the database object.
- You have the SHOW component privilege for the SQL_OPERATIONS component. The SHOW component privilege is granted to PUBLIC by default.
- You have the SELECT privilege on the target object.

Differences Between SHOWDDL Output and Original DDL

- All column constraints (NOT NULL, PRIMARY KEY, and CHECK) are transformed into table constraints. All NOT NULL constraints are consolidated into a single check constraint.
- Check constraints are moved out of the CREATE TABLE statement and encapsulated in a separate ALTER TABLE ADD CONSTRAINT statement.
- SHOWDDL generates ALTER TABLE ADD COLUMN statements for each column that was added to the table.
- All ANSI names in the output are qualified with the schema name.
- SHOWDDL displays constraint names even though they might not have been specified during the creation of the constraint.
- SHOWDDL always generates a Java signature for the SPJ.

PRIVILEGES Option

The PRIVILEGES option includes the GRANT statements as they apply to the option. Each privilege is specified in separate GRANT statements even if they were granted in a single statement.

Examples of SHOWDDL

- This SHOWDDL statement displays the statement that created the specified table in the database and the privileges granted on that table:

```
SQL>showddl tab41;


CREATE TABLE TRAFODION.SCH41.TAB41
  (
    A                               INT DEFAULT NULL
  , B                               INT DEFAULT NULL
  )
;

-- GRANT DELETE, INSERT, SELECT, UPDATE, REFERENCES ON TRAFODION."SCH41"."TAB41" TO PAULLOW41 WITH GRANT
 OPTION;

--- SQL operation complete.
```

- This SHOWDDL statement displays the statement that registered the specified user in the database:

```
SQL>showddl user sqluser_admin;


REGISTER USER "SQLUSER_ADMIN";

--- SQL operation complete.
```

- This SHOWDDL statement displays the statement that created the specified role in the database and the users who have been granted this role:

```
SQL>showddl role db__rootrole;


CREATE ROLE "DB__ROOTROLE";
```

```
  -- GRANT ROLE "DB__ROOTROLE" TO "DB__ROOT" WITH ADMIN OPTION;

--- SQL operation complete.
```

SHOWDDL SCHEMA Statement

- "Syntax Description for SHOWDDL SCHEMA"
- "Considerations for SHOWDDL SCHEMA"
- "Example of SHOWDDL SCHEMA"

The SHOWDDL SCHEMA statement displays the DDL syntax used to create a schema as it exists in the metadata and shows the authorization ID that owns the schema.

SHOWDDL SCHEMA is a Trafodion SQL extension.

```
SHOWDDL SCHEMA  [catalog-name.]schema-name
```

Syntax Description for SHOWDDL SCHEMA

[*catalog-name*.]*schema-name*

specifies the ANSI name of an existing catalog and schema. If *schema-name* is not fully qualified, SHOWDDL uses the default catalog for the session, TRAFODION. For more information, see "Database Object Names" (page 198).

Considerations for SHOWDDL SCHEMA

If not specified, the catalog is the current default catalog, TRAFODION.

Required Privileges

To issue a SHOWDDL SCHEMA statement, one of the following must be true:

- You are DB__ROOT.
- You are the owner of the schema.
- You have the SHOW component privilege for the SQL_OPERATIONS component. The SHOW component privilege is granted to PUBLIC by default.

Example of SHOWDDL SCHEMA

This SHOWDDL SCHEMA statement displays the DDL syntax used to create the schema, MYSCHEMA, as it exists in the metadata and shows the authorization ID that owns the schema:

```
SHOWDDL SCHEMA MYSCHEMA;

CREATE PRIVATE SCHEMA "TRAFODION"."MYSCHEMA" AUTHORIZATION "DB__ROOT";

--- SQL operation complete.
```

SHOWSTATS Statement

- "Syntax Description of SHOWSTATS"
- "Considerations for SHOWSTATS"
- "Examples of SHOWSTATS"

The SHOWSTATS statement displays the histogram statistics for one or more groups of columns within a table. These statistics are used to devise optimized access plans.

SHOWSTATS is a Trafodion SQL extension.

```
SHOWSTATS FOR TABLE table-name ON group-list [DETAIL]

group-list is:

     column-list[, column-list]...
   | EVERY COLUMN[, column-list]...
   | EVERY KEY[, column-list]...
   | EXISTING COLUMN[S][, column-list]...

 column-list for a single-column group is:

     column-name
   | (column-name)
   | column-name TO column-name
   | (column-name) TO (column-name)
   | column-name TO (column-name)
   | (column-name) TO column-name

 column-list for a multicolumn group is:

    (column-name, column-name[, column-name]...)
```

Syntax Description of SHOWSTATS

table-name

is the ANSI name of the table for which statistics are to be displayed.

ON *group-list*

specifies one or more groups of columns, *group-list*, for which to display histogram statistics.

group-list is: *column-list* [, *column-list*]... | EVERY COLUMN [, *column-list*]... | EVERY KEY [, *column-list*]... | EXISTING COLUMN[S] [, *column-list*]...

specifies the ways in which *group-list* can be defined. The column list represents both a single-column group and a multicolumn group.

EVERY COLUMN

indicates that histogram statistics are to be displayed for each individual column of *table* and any multicolumns that make up the primary key and indexes. For columns that do not have histograms, this option returns `No histogram data for column(s)--->`.

EVERY KEY

indicates that histogram statistics are to be displayed for columns that make up the primary key and indexes.

EXISTING COLUMN[S]

indicates that histogram statistics are to be displayed only for columns of *table* that actually have histograms. This option yields a more concise report because columns with no histogram data are omitted. This option includes any existing multicolumn histograms.

DETAIL

displays statistics for corresponding histogram intervals and other details.

If you do not select the DETAIL keyword, the default display lists the basic histogram information, including the histogram ID, number of intervals, total rows, total UEC, and the column names. The detailed display additionally includes the low value and high value as well as interval data.

column-list for a single-column group is: *column-name* | (*column-name*) | *column-name* TO *column-name* | (*column-name*) TO (*column-name*) | *column-name* TO (*column-name*) | (*column-name*) TO *column-name*

specifies the ways in which the *column-name* can be defined for single-column groups. A range of columns specified using the TO keyword causes all columns in that range to be included, defined by their order of declaration in the table.

column-list for a multicolumn group is: (*column-name, column-name* [, *column-name*]...)

specifies the ways in which the *column-name* can be defined for multicolumn groups. For example, (abc, def) indicates the multicolumn histogram consisting of columns abc and def, not two single-column histograms.

For more information about the column list syntax and specifying columns, see the "UPDATE STATISTICS Statement" (page 186).

Considerations for SHOWSTATS

Required Privileges

To issue a SHOWSTATS statement, one of the following must be true:

- You are DB__ROOT.
- You are the owner of the database object.
- You have the SHOW component privilege for the SQL_OPERATIONS component. The SHOW component privilege is granted to PUBLIC by default.
- You have the SELECT privilege on the target object.
- You have the MANAGE_STATISTICS component privilege for the SQL_OPERATIONS component.

Examples of SHOWSTATS

- This example displays histogram statistics for table A using the EVERY KEY keyword. In addition, the DETAIL keyword is selected:

```
SHOWSTATS FOR TABLE A ON EVERY KEY DETAIL;
```

- This example displays statistics for table CAT.SCH.A and selects all columns from abc through def:

```
SHOWSTATS FOR TABLE CAT.SCH.A ON ABC TO DEF;
```

- This example displays statistics for table A. The list of column names contained within parenthesis refers to a multicolumn group:

```
SHOWSTATS FOR TABLE A ON (ABC,DEF);
```

- This example displays statistics for table A using the EXISTING COLUMNS keyword. In addition, the DETAIL keyword is selected:

```
SHOWSTATS FOR TABLE A ON EXISTING COLUMNS DETAIL;
```

Default output example:

```
>>SHOWSTATS FOR TABLE A ON EXISTING COLUMNS;

Histogram data for Table CAT.SCH.A
Table ID: 341261536378386
```

```
  Hist ID # Ints Rowcount   UEC Colname(s)
========== ====== ========== ==============================
 623327638      1          11          10 ABC, DEF, GHI
 623327633     10          11          10 ABC
 623327628      9          11           9 DEF
 623327623     10          11          10 GHI

--- SQL operation complete.
>>SHOWSTATS FOR TABLE A ON ABC;

Histogram data for Table CAT.SCH.A
Table ID: 341261536378386

  Hist ID # Ints  Rowcount  UEC Colname(s)
========= ====== ========== ==============================
 623327633     10          11          10 ABC

--- SQL operation complete.


>>SHOWSTATS FOR TABLE A ON DEF DETAIL;

Detailed Histogram data for Table CAT.SCH.A
Table ID: 341261536378386

Hist ID:    623327628
Column(s):  DEF
Total Rows: 11
Total UEC:  9
Low Value:  (1)
High Value: (199)
Intervals:  9

Number  Rowcount    UEC Boundary
====== ========== ==============================

     0           0           0 (1)
     1           1           1 (1)
     2           3           1 (2)
     3           1           1 (4)
     4           1           1 (11)
     5           1           1 (12)
     6           1           1 (14)
     7           1           1 (99)
     8           1           1 (123)
     9           1           1 (199)

--- SQL operation complete.
```

TABLE Statement

- "Considerations for TABLE"
- "Example of TABLE"

The TABLE statement is equivalent to the query specification `SELECT * FROM table`.

```
TABLE table
```

table
names the user table or view.

Considerations for TABLE

Relationship to SELECT Statement

The result of the TABLE statement is one form of a simple-table, which refers to the definition of a table reference within a SELECT statement. See the "SELECT Statement" (page 138).

Example of TABLE

This TABLE statement returns the same result as `SELECT * FROM job`:

```
TABLE job;

Job/Code Job Description
-------- --------------------
     100 MANAGER
     200 PRODUCTION SUPV
     250 ASSEMBLER
     300 SALESREP
     400 SYSTEM ANALYST
     420 ENGINEER
     450 PROGRAMMER
     500 ACCOUNTANT
     600 ADMINISTRATOR
     900 SECRETARY

--- 10 row(s) selected.
```

UNREGISTER USER Statement

- "Syntax Description of UNREGISTER USER"
- "Considerations for UNREGISTER USER"
- "Example of UNREGISTER USER"

The UNREGISTER USER statement removes a database username from the SQL database. The user can no longer log on to the database.

UNREGISTER USER is a Trafodion SQL extension.

```
UNREGISTER USER database-username [RESTRICT | CASCADE]
```

Syntax Description of UNREGISTER USER

database-username

is the name of a currently registered database user. *database-username* is a regular or delimited case-insensitive identifier. See "Case-Insensitive Delimited Identifiers" (page 221).

[RESTRICT | CASCADE]

If you specify RESTRICT, the UNREGISTER USER operation fails if there are any objects or schemas in the database owned by the user or any privileges or roles granted to the user.

If you specify CASCADE, all objects and schemas owned by the user are dropped, and all privileges and roles granted to the user are revoked as part of the UNREGISTER USER operation.

The default value is RESTRICT.

Considerations for UNREGISTER USER

- To unregister a user, you must have user administrative privileges. You have user administrative privileges if you have been granted the MANAGE_USERS component privilege. Initially, DB__ROOT is the only database user who has been granted the MANAGE_USERS component privilege.
- You cannot unregister any username beginning with DB__. Role names beginning with DB__ are reserved by Trafodion.
- UNREGISTER USER fails if you specify RESTRICT (or nothing) and if the user owns any objects or schemas or if the user has been granted any privileges or roles.

Example of UNREGISTER USER

To unregister a user:

```
UNREGISTER USER "jsmith@hp.com";
```

UPDATE Statement

- "Syntax Description of UPDATE"
- "Considerations for UPDATE"
- "Examples of UPDATE"

The UPDATE statement is a DML statement that updates data in a row or rows in a table or updatable view. Updating rows in a view updates the rows in the table on which the view is based.

```
Searched UPDATE is:

UPDATE table

 { set-clause-type1 | set-clause-type2 }

   set-clause-type1 is:
      SET set-clause[, set-clause ]..
      set-clause is:
         column-name = {expression | NULL}

   set-clause-type2 is:
      SET (column1, ..., columnN) = {(value1, ..., valueN) | (query-expr)}

   [WHERE search-condition]
   [[FOR] access-option ACCESS]

access-option is:
   READ COMMITTED
```

Syntax Description of UPDATE

table

names the user table or view to update. *table* must be a base table or an updatable view. To refer to a table or view, use the ANSI logical name.

See "Database Object Names" (page 198).

set-clause-type1

This type of SET clause associates a value with a specific column in the table being updated. For each *set-clause*, the value of the specified target *column-name* is replaced by the value of the update source *expression* (or NULL). The data type of each target column must be compatible with the data type of its source value.

column-name

names a column in *table* to update. You cannot qualify or repeat a column name. You cannot update the value of a column that is part of the primary key.

expression

is an SQL value expression that specifies a value for the column. The *expression* cannot contain an aggregate function defined on a column. The data type of *expression* must be compatible with the data type of *column-name*.

If *expression* refers to columns being updated, Trafodion SQL uses the original values to evaluate the expression and determine the new value.

See "Expressions" (page 211).

NULL

can also specify the value of the update source.

set-clause-type2

This type of SET clause allows multiple columns to be specified on the left side of the assignment operator. These columns are updated using multiple values specified on the right side of the

assignment operator. The right side of the assignment operator could be simple values or a subquery.

column1, ..., *columnN*

names columns in *table* to update. You cannot qualify or repeat a column name. You cannot update the value of a column that is part of the primary key.

value1, ..., *valueN*

are values specified on the right side of the assignment operator for the columns specified on the left side of the assignment operator. The data type of each value must be compatible with the data type of the corresponding column on the left side of the assignment operator.

query-expr

is a SELECT subquery. Only one subquery can be specified on the right side of a SET clause. The subquery cannot refer to the table being updated. For the syntax and description of *query-expr*, see the "SELECT Statement" (page 138).

WHERE *search-condition*

specifies a *search-condition* that selects rows to update. Within the *search-condition*, columns being compared are also being updated in the table or view. See "Search Condition" (page 250).

If you do not specify a *search-condition*, all rows in the table or view are updated.

Do not use an UPDATE statement with a WHERE clause that contains a SELECT for the same table. Reading from and inserting into, updating in, or deleting from the same table generates an error. Use a positioned (WHERE CURRENT OF) UPDATE instead. See "MERGE Statement" (page 123).

[FOR] *access-option* ACCESS

specifies the *access-option* required for data used in the evaluation of a search condition. See "Data Consistency and Access Options" (page 25).

READ COMMITTED

specifies that any data used in the evaluation of the search condition must be from committed rows.

Considerations for UPDATE

Performance

An UPDATE of primary key columns could perform poorly when compared to an UPDATE of non-key columns. This is because the UPDATE operation involves moving records in disk by deleting all the records in the before-image and then inserting the records in the after-image back into the table.

Authorization Requirements

UPDATE requires authority to read and write to the table or view being updated and authority to read any table or view specified in subqueries used in the search condition. A column of a view can be updated if its underlying column in the base table can be updated.

Transaction Initiation and Termination

The UPDATE statement automatically initiates a transaction if no active transaction exists. Otherwise, you can explicitly initiate a transaction with the BEGIN WORK statement. When a transaction is started, the SQL statements execute within that transaction until a COMMIT or ROLLBACK is encountered or an error occurs.

Isolation Levels of Transactions and Access Options of Statements

The isolation level of a Trafodion SQL transaction defines the degree to which the operations on data within that transaction are affected by operations of concurrent transactions. When you specify

access options for the DML statements within a transaction, you override the isolation level of the containing transaction. Each statement then executes with its individual access option.

Conflicting Updates in Concurrent Applications

If you are using the READ COMMITTED isolation level within a transaction, your application can read different committed values for the same data at different times. Further, two concurrent applications can update (possibly in error) the same column in the same row.

Requirements for Data in Row

Each row to be updated must satisfy the constraints of the table or underlying base table of the view. No column updates can occur unless all of these constraints are satisfied. (A table constraint is satisfied if the check condition is not false—that is, it is either true or has an unknown value.)

In addition, a candidate row from a view created with the WITH CHECK OPTION must satisfy the view selection criteria. The selection criteria are specified in the WHERE clause of the AS *query-expr* clause in the CREATE VIEW statement.

Reporting of Updates

When an UPDATE completes successfully, Trafodion SQL reports the number of times rows were updated during the operation.

Under certain conditions, updating a table with indexes can cause Trafodion SQL to update the same row more than once, causing the number of reported updates to be higher than the actual number of changed rows. However, both the data in the table and the number of reported updates are correct. This behavior occurs when all of these conditions are true:

- The optimizer chooses an alternate index as the access path.
- The index columns specified in WHERE *search-condition* are not changed by the update.
- Another column within the same index is updated to a higher value (if that column is stored in ascending order), or a lower value (if that column is stored in descending order).

When these conditions occur, the order of the index entries ensures that Trafodion SQL will encounter the same row (satisfying the same *search-condition)* at a later time during the processing of the table. The row is then updated again by using the same value or values.

For example, suppose that the index of MYTABLE consists of columns A and B, and the UPDATE statement is specified:

```
UPDATE MYTABLE
SET B = 20
WHERE A > 10;
```

If the contents of columns A and B are 11 and 12 respectively before the UPDATE, after the UPDATE Trafodion SQL will encounter the same row indexed by the values 11 and 20.

Updating Character Values

For a fixed-length character column, an update value shorter than the column length is padded with single-byte ASCII blanks (HEX20) to fill the column. If the update value is longer than the column length, string truncation of non blank trailing characters returns an error, and the column is not updated.

For a variable-length character column, an update value is not padded; its length is the length of the value specified. As is the case for fixed length, if the update value is longer than the column length, string truncation of non blank trailing characters returns an error, and the column is not updated.

SET Clause Restrictions and Error Cases

The SET clause has the following restrictions:

- The number of columns on the left side of each assignment operator should match the number of values or SELECT list elements on the right side. The following examples are **not** allowed:

```
UPDATE t SET (a,b)=(10,20,30)
UPDATE t set (b,c)=(SELECT r,t,s FROM x)
```

- If multi-column update syntax is specified and the right side contains a subquery, only one element, the subquery, is not allowed.

```
UPDATE t SET (a,b)=(10, (SELECT a FROM t1))
```

- More than one subquery is not allowed if multiple-column syntax is used.

```
UPDATE t SET (a,b)=(SELECT x,y FROM z), (c,d)=(SELECT x,y FROM a))
```

- If a subquery is used, it must return at most one row.

Examples of UPDATE

- Update a single row of the ORDERS table that contains information about order number 200300 and change the delivery date:

```
UPDATE sales.orders
SET deliv_date = DATE '2008-05-02'
WHERE ordernum = 200300;
```

- Update several rows of the CUSTOMER table:

```
UPDATE sales.customer
SET credit = 'A1'
WHERE custnum IN (21, 3333, 324);
```

- Update all rows of the CUSTOMER table to the default credit 'C1':

```
UPDATE sales.customer
SET credit = 'C1';
```

- Update the salary of each employee working for all departments located in Chicago:

```
UPDATE persnl.employee
SET salary = salary * 1.1
WHERE deptnum IN
  (SELECT deptnum FROM persnl.dept
   WHERE location = 'CHICAGO');
```

  The subquery is evaluated for each row of the DEPT table and returns department numbers for departments located in Chicago.

- This is an example of a self-referencing UPDATE statement, where the table being updated is scanned in a subquery:

```
UPDATE table3 SET b = b + 2000 WHERE a, b =
(SELECT a, b FROM table3 WHERE b > 200);
```

UPSERT Statement

- "Syntax Description of UPSERT"
- "Examples of UPSERT"

The UPSERT statement either updates a table if the row exists or inserts into a table if the row does not exist.

UPSERT is a Trafodion SQL extension.

```
UPSERT [USING LOAD] INTO table [(target-col-list)] {query-expr | values-clause}

target-col-list is:
     column-name[, column-name]...

values-clause is:
     VALUES ( expression[, expression]... )
```

Syntax Description of UPSERT

USING LOAD

allows the UPSERT to occur without a transaction. Use this clause when inserting data into an empty table. If you do not specify this clause, the UPSERT occurs within a transaction.

table

names the user table in which to insert or update rows. *table* must be a base table.

(*target-col-list*)

names the columns in the table in which to insert or update values. The data type of each target column must be compatible with the data type of its corresponding source value. Within the list, each target column must have the same position as its associated source value, whose position is determined by the columns in the table derived from the evaluation of the query expression (*query-expr*).

If you do not specify all of the columns in the target *table* in the *target-col-list*, column default values are inserted into or updated in the columns that do not appear in the list. See "Column Default Settings" (page 194).

If you do not specify *target-col-list*, row values from the source table are inserted into or updated in all columns in table. The order of the column values in the source table must be the same order as that of the columns specified in the CREATE TABLE for *table*. (This order is the same as that of the columns listed in the result table of SHOWDDL *table*.)

column-name

names a column in the target *table* in which to either insert or update data. You cannot qualify or repeat a column name.

query-expr

is a SELECT subquery that returns data to be inserted into or updated in the target *table*. The subquery cannot refer to the table being operated on. For the syntax and description of *query-expr*, see the "SELECT Statement" (page 138).

VALUES (*expression*[, *expression*]...)

specifies an SQL value expression or a set of expressions that specify values to be inserted into or updated in the target *table*. The data type of *expression* must be compatible with the data type of the corresponding column in the target *table*. See "Expressions" (page 211).

Examples of UPSERT

- This UPSERT statement either inserts or updates the part number and price in the PARTS table using the part number and unit price from the ODETAIL table where the part number is 244:

```
UPSERT INTO sales.parts (partnum, price) SELECT partnum, unit_price
   FROM sales.odetail WHERE partnum = 244;
```

- This UPSERT statement either inserts or updates rows in the EMPLOYEE table using the results of querying the EMPLOYEE_EUROPE table:

  ```
  UPSERT INTO persnl.employee SELECT * FROM persnl.employee_europe;
  ```

- This UPSERT statement either inserts or updates a row in the DEPT table using the specified values:

  ```
  UPSERT INTO persnl.dept VALUES (3500,'CHINA SALES',111,3000,'HONG KONG');
  ```

- This UPSERT statement either inserts or updates a row in the DEPT table using the specified values:

  ```
  UPSERT INTO persnl.dept (deptnum, deptname, manager)
     VALUES (3600,‘JAPAN SALES’, 996);
  ```

VALUES Statement

- "Considerations for VALUES"
- "Examples of VALUES"

The VALUES statements starts with the VALUES keyword followed by a sequence of row value constructors, each of which is enclosed in parenthesis. It displays the results of the evaluation of the expressions and the results of row subqueries within the row value constructors.

```
VALUES (row-value-const) [, (row-value-const)]...

row-value-const is:
     row-subquery
     | {expression | NULL} [,{expression | NULL}...
```

row-value-const

specifies a list of expressions (or NULL) or a row subquery (a subquery that returns a single row of column values). An operand of an expression cannot reference a column (except when the operand is a scalar subquery returning a single column value in its result table).

The results of the evaluation of the expressions and the results of the row subqueries in the row value constructors must have compatible data types.

Considerations for VALUES

Relationship to SELECT Statement

The result of the VALUES statement is one form of a *simple-table*, which is part of the definition of a table reference within a SELECT statement. See the "SELECT Statement" (page 138).

Relationship to INSERT Statement

For a VALUES clause that is the direct source of an INSERT statement, Trafodion SQL also allows the keyword DEFAULT in a VALUES clause, just like NULL is allowed. For more information, see the "INSERT Statement" (page 118).

Examples of VALUES

- This VALUES statement displays two rows with simple constants:

```
VALUES (1,2,3), (4,5,6);

(EXPR)  (EXPR)   (EXPR)
------   ------   -----
     1        2       3
     4        5       6

--- 2 row(s) selected.
```

- This VALUES statement displays the results of the expressions and the row subquery in the lists:

```
VALUES (1+2, 3+4), (5, (select count (*) from t));

(EXPR)   (EXPR)
------   ----------------- ------
     3                        7
     5                        2

--- 2 row(s) selected.
```

3 SQL Utilities

A utility is a tool that runs within Trafodion SQL and performs tasks. This section describes the Trafodion SQL utilities:

"LOAD Statement" (page 177)	Uses the Trafodion Bulk Loader to load data from a source table, either a Trafodion table or a Hive table, into a target Trafodion table.
"POPULATE INDEX Utility" (page 180)	Loads indexes.
"PURGEDATA Utility" (page 182)	Purges data from tables and indexes.
"UNLOAD Statement" (page 183)	Unloads data from Trafodion tables into an HDFS location that you specify.
"UPDATE STATISTICS Statement" (page 186)	Updates the histogram statistics for one or more groups of columns within a table. These statistics are used to devise optimized access plans.

NOTE: Trafodion SQL utilities are entered interactively or from script files using a client-based tool, such as the Trafodion Command Interface (TrafCI). To install and configure a client application that enables you to connect to and issue SQL utilities, see the *Trafodion Client Installation Guide*.

LOAD Statement

- "Syntax Description of LOAD"
- "Considerations for LOAD"
- "Example of LOAD"

The LOAD statement uses the Trafodion Bulk Loader to load data from a source table, either a Trafodion table or a Hive table, into a target Trafodion table. The Trafodion Bulk Loader prepares and loads HFiles directly in the region servers and bypasses the write path and the cost associated with it. The write path begins at a client, moves to a region server, and ends when data eventually is written to an HBase data file called an HFile.

The Trafodion bulk load process takes place in two phases:

- *Preparation phase*: In this phase, Trafodion reads the data from the source files in Hive or HDFS, partitions the data based on the target table's partitioning scheme, sorts the data, and then generates KeyValue pairs that will populate the HFiles. Trafodion also encodes the data for faster storage and retrieval.
- *Loading-the-files-into-HBase phase*: This phase uses the LoadIncrementalHFiles (also known as the computebulkload tool) and load the generated HFiles into the region servers.

LOAD is a Trafodion SQL extension.

```
LOAD [WITH option[[,] option]...] INTO target-table SELECT ... FROM source-table

option is:
     TRUNCATE TABLE
   | NO RECOVERY
   | NO POPULATE INDEXES
   | NO DUPLICATE CHECK
   | NO OUTPUT
   | INDEX TABLE ONLY
   | UPSERT USING LOAD
```

Syntax Description of LOAD

target-table

is the name of the target Trafodion table where the data will be loaded. See "Database Object Names" (page 198).

source-table

is the name of either a Trafodion table or a Hive table that has the source data. Hive tables can be accessed in Trafodion using the HIVE.HIVE schema (for example, hive.hive.orders). The Hive table needs to already exist in Hive before Trafodion can access it. If you want to load data that is already in an HDFS folder, then you need to create an external Hive table with the right fields and pointing to the HDFS folder containing the data. You can also specify a WHERE clause on the source data as a filter.

[WITH *option*[[,] *option*]...]

is a set of options that you can specify for the load operation. You can specify one or more of these options:

TRUNCATE TABLE

causes the Bulk Loader to truncate the target table before starting the load operation. By default, the Bulk Loader does not truncate the target table before loading data.

NO RECOVERY

specifies that the Bulk Loader not use HBase snapshots for recovery. By default, the Bulk Loader handles recovery using the HBase snapshots mechanism.

NO POPULATE INDEXES

specifies that the Bulk Loader not handle index maintenance or populate the indexes. By default, the Bulk Loader handles index maintenance, disabling indexes before starting the load operation and populating them after the load operation is complete.

NO DUPLICATE CHECK

causes the Bulk Loader to ignore duplicates in the source data. By default, the Bulk Loader checks if there are duplicates in the source data and generates an error when it detects duplicates.

NO OUTPUT

prevents the LOAD statement from displaying status messages. By default, the LOAD statement prints status messages listing the steps that the Bulk Loader is executing.

INDEX TABLE ONLY

specifies that the target table, which is an index, be populated with data from the parent table.

UPSERT USING LOAD

specifies that the data be inserted into the target table using rowset inserts without a transaction.

Considerations for LOAD

Required Privileges

To issue a LOAD statement, one of the following must be true:

- You are DB__ROOT.
- You are the owner of the target table.
- You have these privileges:
  - SELECT and INSERT privileges on the target table
  - DELETE privilege on the target table if TRUNCATE TABLE is specified
- You have the MANAGE_LOAD component privilege for the SQL_OPERATIONS component.

Configuration Before Running LOAD

Before running the LOAD statement, make sure that you have configured the staging folder, source table, and HBase according to these guidelines.

Staging Folder for HFiles

The Bulk Loader uses an HDFS folder as a staging area for the HFiles before calling HBase APIs to merge them into the Trafodion table. By default, Trafodion uses `/bulkload/` as the staging folder. This folder must be owned by the same user as the one under which Trafodion runs. Trafodion also must have full permissions on this folder. The HBase user (that is, the user under which HBase runs) must have read/write access to this folder.

Example:

```
drwxr-xr-x - trafodion trafodion 0 2014-07-07 09:49 /bulkload.
```

Hive Source Table

To load data stored in HDFS, you will need to create a Hive table with the right fields and types pointing to the HDFS folder containing the data before you start the load.

HBase Snapshots

If you do not specify the NO RECOVERY OPTION in the LOAD statement, the Bulk Loader uses HBase snapshots as a mechanism for recovery. Snapshots are a lightweight operation where some metadata is copied. (Data is not copied.) A snapshot is taken before the load starts and is removed after the load completes successfully. If something goes wrong and it is possible to recover, the snapshot is used to restore the table to its initial state before the load started. To use this recovery mechanism, HBase needs to be configured to allow snapshots.

Example of LOAD

- For customer demographics data residing in `/hive/tpcds/customer_demographics`, create an external Hive table using the following Hive SQL:

```
create external table customer_demographics
(
    cd_demo_sk int,
    cd_gender string,
    cd_marital_status string,
    cd_education_status string,
    cd_purchase_estimate int,
    cd_credit_rating string,
    cd_dep_count int,
    cd_dep_employed_count int,
    cd_dep_college_count int
)
row format delimited fields terminated by '|'
location '/hive/tpcds/customer_demographics';
```

- The Trafodion table where you want to load the data is defined using this DDL:

```
create table customer_demographics_salt
(
  cd_demo_sk int not null,
  cd_gender char(1),
  cd_marital_status char(1),
  cd_education_status char(20),
  cd_purchase_estimate int,
  cd_credit_rating char(10),
  cd_dep_count int,
  cd_dep_employed_count int,
  cd_dep_college_count int,
  primary key (cd_demo_sk)
)
salt using 4 partitions on (cd_demo_sk);
```

- This example shows how the LOAD statement loads the customer_demographics_salt table from the Hive table, hive.hive.customer_demographics:

```
>>load into customer_demographics_salt
+>select * from hive.hive.customer_demographics where cd_demo_sk <= 5000;
Task: LOAD Status: Started Object: TRAFODION.HBASE.CUSTOMER_DEMOGRAPHICS_SALT
Task: DISABLE INDEX Status: Started Object: TRAFODION.HBASE.CUSTOMER_DEMOGRAPHICS_SALT
Task: DISABLE INDEX Status: Ended Object: TRAFODION.HBASE.CUSTOMER_DEMOGRAPHICS_SALT
Task: PREPARATION Status: Started Object: TRAFODION.HBASE.CUSTOMER_DEMOGRAPHICS_SALT
       Rows Processed: 5000
Task: PREPARATION Status: Ended ET: 00:00:03.199
Task: COMPLETION Status: Started Object: TRAFODION.HBASE.CUSTOMER_DEMOGRAPHICS_SALT
Task: COMPLETION Status: Ended ET: 00:00:00.331
Task: POPULATE INDEX Status: Started Object: TRAFODION.HBASE.CUSTOMER_DEMOGRAPHICS_SALT
Task: POPULATE INDEX Status: Ended ET: 00:00:05.262
```

POPULATE INDEX Utility

- "Syntax Description of POPULATE INDEX"
- "Considerations for POPULATE INDEX"
- "Examples of POPULATE INDEX"

The POPULATE INDEX utility performs a fast INSERT of data into an index from the parent table. You can execute this utility in a client-based tool like TrafCI.

```
POPULATE INDEX index ON table [index-option]

index-option is:
   ONLINE | OFFLINE
```

Syntax Description of POPULATE INDEX

index

is an SQL identifier that specifies the simple name for the index. You cannot qualify *index* with its schema name. Indexes have their own namespace within a schema, so an index name might be the same as a table or constraint name. However, no two indexes in a schema can have the same name.

table

is the name of the table for which to populate the index. See "Database Object Names" (page 198).

ONLINE

specifies that the populate operation should be done online. That is, ONLINE allows read and write DML access on the base table while the populate operation occurs. Additionally, ONLINE reads the audit trail to replay updates to the base table during the populate phase. If a lot of audit is generated and you perform many CREATE INDEX operations, we recommend that you avoid ONLINE operations because they can add more contention to the audit trail. The default is ONLINE.

OFFLINE

specifies that the populate should be done offline. OFFLINE allows only read DML access to the base table. The base table is unavailable for write operations at this time. OFFLINE must be specified explicitly. SELECT is allowed.

Considerations for POPULATE INDEX

When POPULATE INDEX is executed, the following steps occur:

- The POPULATE INDEX operation runs in many transactions.
- The actual data load operation is run outside of a transaction.

If a failure occurs, the rollback is faster because it does not have to process a lot of audit. Also, if a failure occurs, the index remains empty, unaudited, and not attached to the base table (offline).

- When an offline POPULATE INDEX is being executed, the base table is accessible for read DML operations. When an online POPULATE INDEX is being executed, the base table is accessible for read and write DML operations during that time period, except during the commit phase at the very end.
- If the POPULATE INDEX operation fails unexpectedly, you may need to drop the index again and re-create and repopulate.
- Online POPULATE INDEX reads the audit trail to replay updates by allowing read/write access. If you plan to create many indexes in parallel or if you have a high level of activity on the audit trail, you should consider using the OFFLINE option.

Errors can occur if the source base table or target index cannot be accessed, or if the load fails due to some resource problem or problem in the file system.

Required Privileges

To perform a POPULATE INDEX operation, one of the following must be true:

- You are DB__ROOT.
- You are the owner of the table.
- You have the SELECT and INSERT (or ALL) privileges on the associated table.

Examples of POPULATE INDEX

- This example loads the specified index from the specified table:

```
POPULATE INDEX myindex ON myschema.mytable;
```

- This example loads the specified index from the specified table, which uses the default schema:

```
POPULATE INDEX index2 ON table2;
```

PURGEDATA Utility

- "Syntax Description of PURGEDATA"
- "Considerations for PURGEDATA"
- "Example of PURGEDATA"

The PURGEDATA utility performs a fast DELETE of data from a table and its related indexes. You can execute this utility in a client-based tool like TrafCI.

```
PURGEDATA object
```

Syntax Description of PURGEDATA

object

is the name of the table from which to purge the data. See "Database Object Names" (page 198).

Considerations for PURGEDATA

- The *object* can be a table name.
- Errors are returned if *table* cannot be accessed or if a resource or file-system problem causes the delete to fail.
- PURGEDATA is not supported for volatile tables.

Required Privileges

To perform a PURGEDATA operation, one of the following must be true:

- You are DB__ROOT.
- You are the owner of the table.
- You have the SELECT and DELETE (or ALL) privileges on the associated table.

Availability

PURGEDATA marks the table OFFLINE and sets the corrupt bit while processing. If PURGEDATA fails before it completes, the table and its dependent indexes will be unavailable, and you must run PURGEDATA again to complete the operation and remove the data. Error 8551 with an accompanying file system error 59 or error 1071 is returned in this case.

Example of PURGEDATA

This example purges the data in the specified table. If the table has indexes, their data is also purged.

```
PURGEDATA myschema.mytable;
```

UNLOAD Statement

- "Syntax Description of UNLOAD"
- "Considerations for UNLOAD"
- "Example of UNLOAD"

The UNLOAD statement unloads data from Trafodion tables into an HDFS location that you specify. Extracted data can be either compressed or uncompressed based on what you choose.

UNLOAD is a Trafodion SQL extension.

```
UNLOAD [WITH option[ option]...] INTO 'target-location' SELECT ... FROM source-table ...

option is:
     DELIMITER { 'delimiter-string' | delimiter-ascii-value }
   | RECORD_SEPARATOR { 'separator-literal' | separator-ascii-value }
   | NULL_STRING 'string-literal'
   | PURGEDATA FROM TARGET
   | COMPRESSION GZIP
   | MERGE FILE merged_file-path [OVERWRITE]
   | NO OUTPUT
   | { NEW | EXISTING } SNAPSHOT HAVING SUFFIX 'string'
```

Syntax Description of UNLOAD

'*target-location*'

is the full pathname of the target HDFS folder where the extracted data will be written. Enclose the name of folder in single quotes. Specify the folder name as a full pathname and not as a relative path. You must have write permissions on the target HDFS folder. If you run UNLOAD in parallel, multiple files will be produced under the *target-location*. The number of files created will equal the number of ESPs.

SELECT ... FROM *source-table* ...

is either a simple query or a complex one that contains GROUP BY, JOIN, or UNION clauses. *source-table* is the name of a Trafodion table that has the source data. See "Database Object Names" (page 198).

[WITH *option*[*option*]...]

is a set of options that you can specify for the unload operation. If you specify an option more than once, Trafodion returns an error with SQLCODE -4489. You can specify one or more of these options:

DELIMITER { '*delimiter-string*' | *delimiter-ascii-value* }

specifies the delimiter as either a delimiter string or an ASCII value. If you do not specify this option, Trafodion uses the character "|" as the delimiter.

delimiter-string can be any ASCII or Unicode string. You can also specify the delimiter as an ASCII value. Valid values range from 1 to 255. Specify the value in decimal notation; hexadecimal or octal notation are currently not supported. If you are using an ASCII value, the delimiter can be only one character wide. Do not use quotes when specifying an ASCII value for the delimiter.

RECORD_SEPARATOR { '*separator-literal*' | *separator-ascii-value* }

specifies the character that will be used to separate consecutive records or rows in the output file. You can specify either a literal or an ASCII value for the separator. The default value is a newline character.

separator-literal can be any ASCII or Unicode character. You can also specify the separator as an ASCII value. Valid values range from 1 to 255. Specify the value in decimal notation; hexadecimal or octal notation are currently not supported. If you are using an ASCII value, the separator can be only one character wide. Do not use quotes when specifying an ASCII value for the separator.

NULL_STRING '*string-literal*'

specifies the string that will be used to indicate a NULL value. The default value is the empty string ''.

PURGEDATA FROM TARGET

causes files in the target HDFS folder to be deleted before the unload operation.

COMPRESSION GZIP

uses gzip compression in the extract node, writing the data to disk in this compressed format. GZIP is currently the only supported type of compression. If you do not specify this option, the extracted data will be uncompressed.

MERGE FILE *merged_file-path* [OVERWRITE]

merges the unloaded files into one single file in the specified *merged-file-path*. If you specify compression, the unloaded data will be in compressed format, and the merged file will also be in compressed format. If you specify the optional OVERWRITE keyword, the file is overwritten if it already exists; otherwise, Trafodion raises an error if the file already exists.

NO OUTPUT

prevents the UNLOAD statement from displaying status messages. By default, the UNLOAD statement prints status messages listing the steps that the Bulk Unloader is executing.

{ NEW | EXISTING } SNAPSHOT HAVING SUFFIX '*string*'

initiates an HBase snapshot scan during the unload operation. During a snapshot scan, the Bulk Unloader will get a list of the Trafodion tables from the query explain plan and will create and verify snapshots for the tables. Specify a suffix string, '*string*', which will be appended to each table name.

Considerations for UNLOAD

- You must have write permissions on the target HDFS folder.
- If a WITH option is specified more than once, Trafodion returns an error with SQLCODE -4489.

Required Privileges

To issue an UNLOAD statement, one of the following must be true:

- You are DB__ROOT.
- You are the owner of the target table.
- You have the SELECT privilege on the target table.
- You have the MANAGE_LOAD component privilege for the SQL_OPERATIONS component.

Example of UNLOAD

This example shows how the UNLOAD statement extracts data from a Trafodion table, TRAFODION.HBASE.CUSTOMER_DEMOGRAPHICS, into an HDFS folder, `/bulkload/customer_demographics`:

```
>>UNLOAD
+>WITH PURGEDATA FROM TARGET
+>MERGE FILE 'merged_customer_demogs.gz' OVERWRITE
+>COMPRESSION GZIP
+>INTO '/bulkload/customer_demographics'
+>select * from trafodion.hbase.customer_demographics
+><<+ cardinality 10e10 >>;
Task: UNLOAD Status: Started
Task: EMPTY TARGET Status: Started
Task: EMPTY TARGET Status: Ended ET: 00:00:00.014
```

```
Task: EXTRACT Status: Started
       Rows Processed: 200000
Task: EXTRACT Status: Ended ET: 00:00:04.743
Task: MERGE FILES Status: Started
Task: MERGE FILES Status: Ended ET: 00:00:00.063

--- 200000 row(s) unloaded.
```

UPDATE STATISTICS Statement

- "Syntax Description of UPDATE STATISTICS"
- "Considerations for UPDATE STATISTICS"
- "Examples of UPDATE STATISTICS"

The UPDATE STATISTICS statement updates the histogram statistics for one or more groups of columns within a table. These statistics are used to devise optimized access plans.

UPDATE STATISTICS is a Trafodion SQL extension.

```
UPDATE STATISTICS FOR TABLE table [CLEAR | on-clause]

on-clause is:
     ON column-group-list CLEAR
   | ON column-group-list [histogram-option]...

column-group-list is:
     column-list [,column-list]...
   | EVERY COLUMN [,column-list]...
   | EVERY KEY [,column-list]...
   | EXISTING COLUMN[S] [,column-list]...
   | NECESSARY COLUMN[S] [,column-list]...

column-list for a single-column group is:
     column-name
   | (column-name)
   | column-name TO column-name
   | (column-name) TO (column-name)
   | column-name TO (column-name)
   | (column-name) TO column-name

column-list for a multicolumn group is:
     (column-name, column-name [,column-name]...)

histogram-option is:
     GENERATE n INTERVALS
   | SAMPLE [sample-option]

sample-option is:
     [r ROWS]
   | RANDOM percent PERCENT
   | PERIODIC size ROWS EVERY period ROWS
```

Syntax Description of UPDATE STATISTICS

table

names the table for which statistics are to be updated. To refer to a table, use the ANSI logical name.

See "Database Object Names" (page 198).

CLEAR

deletes some or all histograms for the table *table*. Use this option when new applications no longer use certain histogram statistics.

If you do not specify *column-group-list*, all histograms for *table* are deleted.

If you specify *column-group-list*, only columns in the group list are deleted.

ON *column-group-list*

specifies one or more groups of columns for which to generate histogram statistics with the option of clearing the histogram statistics. You must use the ON clause to generate statistics stored in histogram tables.

column-list

specifies how *column-group-list* can be defined. The column list represents both a single-column group and a multi-column group.

Single-column group:

column-name | (*column-name*) | *column-name* TO *column-name* | (*column-name*) TO (*column-name*)

specifies how you can specify individual columns or a group of individual columns.

To generate statistics for individual columns, list each column. You can list each single column name within or without parentheses.

Multi-column group:

(*column-name*, *column-name* [,*column-name*]...)

specifies a multi-column group.

To generate multi-column statistics, group a set of columns within parentheses, as shown. You cannot specify the name of a column more than once in the same group of columns.

One histogram is generated for each unique column group. Duplicate groups, meaning any permutation of the same group of columns, are ignored and processing continues. When you run UPDATE STATISTICS again for the same user table, the new data for that table replaces the data previously generated and stored in the table's histogram tables. Histograms of column groups not specified in the ON clause remain unchanged in histogram tables.

For more information about specifying columns, see "Generating and Clearing Statistics for Columns" (page 190).

EVERY COLUMN

The EVERY COLUMN keyword indicates that histogram statistics are to be generated for each individual column of *table* and any multicolumns that make up the primary key and indexes. For example, *table* has columns A, B, C, D defined, where A, B, C compose the primary key. In this case, the ON EVERY COLUMN option generates a single column histogram for columns A, B, C, D, and two multi-column histograms of (A, B, C) and (A, B).

The EVERY COLUMN option does what EVERY KEY does, with additional statistics on the individual columns.

EVERY KEY

The EVERY KEY keyword indicates that histogram statistics are to be generated for columns that make up the primary key and indexes. For example, *table* has columns A, B, C, D defined. If the primary key comprises columns A, B, statistics are generated for (A, B), A and B. If the primary key comprises columns A, B, C, statistics are generated for (A,B,C), (A,B), A, B, C. If the primary key comprises columns A, B, C, D, statistics are generated for (A, B, C, D), (A, B, C), (A, B), and A, B, C, D.

EXISTING COLUMN[S]

The EXISTING COLUMN keyword indicates that all existing histograms of the table are to be updated. Statistics must be previously captured to establish existing columns.

NECESSARY COLUMN[S]

The NECESSARY COLUMN[S] keyword generates statistics for histograms that the optimizer has requested but do not exist. Update statistics automation must be enabled for NECESSARY COLUMN[S] to generate statistics. To enable automation, see "Automating Update Statistics" (page 190).

histogram-option

GENERATE *n* INTERVALS

The GENERATE *n* INTERVALS option for UPDATE STATISTICS accepts values between 1 and 10,000. Keep in mind that increasing the number of intervals per histograms may have a negative impact on compile time.

Increasing the number of intervals can be used for columns with small set of possible values and large variance of the frequency of these values. For example, consider a column 'CITY' in table SALES, which stores the city code where the item was sold, where number of cities in the sales data is 1538. Setting the number of intervals to a number greater or equal to the number of cities (that is, setting the number of intervals to 1600) guarantees that the generated histogram captures the number of rows for each city. If the specified value n exceeds the number of unique values in the column, the system generates only as many intervals as the number of unique values.

SAMPLE [*sample-option*]

is a clause that specifies that sampling is to be used to gather a subset of the data from the table. UPDATE STATISTICS stores the sample results and generates histograms.

If you specify the SAMPLE clause without additional options, the result depends on the number of rows in the table. If the table contains no more than 10,000 rows, the entire table will be read (no sampling). If the number of rows is greater than 10,000 but less than 1 million, 10,000 rows are randomly sampled from the table. If there are more than 1 million rows in the table, a random row sample is used to read 1 percent of the rows in the table, with a maximum of 1 million rows sampled.

TIP: As a guideline, the default sample of 1 percent of the rows in the table, with a maximum of 1 million rows, provides good statistics for the optimizer to generate good plans.

If you do not specify the SAMPLE clause, if the table has fewer rows than specified, or if the sample size is greater than the system limit, Trafodion SQL reads all rows from *table*.

See "SAMPLE Clause" (page 261).

sample-option

r ROWS

A row sample is used to read *r* rows from the table. The value *r* must be an integer that is greater than zero (r > 0).

RANDOM *percent* PERCENT

directs Trafodion SQL to choose rows randomly from the table. The value percent must be a value between zero and 100 (0 < percent <= 100). In addition, only the first four digits to the right of the decimal point are significant. For example, value 0.00001 is considered to be 0.0000, Value 1.23456 is considered to be 1.2345.

PERIODIC *size* ROWS EVERY *period* ROW

directs Trafodion SQL to choose the first *size* number of rows from each *period* of rows. The value *size* must be an integer that is greater than zero and less than or equal to the value *period*. (0 < *size* <= *period*). The size of the *period* is defined by the number of rows specified for *period*. The value *period* must be an integer that is greater than zero (*period* > 0).

Considerations for UPDATE STATISTICS

Using Statistics

Use UPDATE STATISTICS to collect and save statistics on columns. The SQL compiler uses histogram statistics to determine the selectivity of predicates, indexes, and tables. Because selectivity directly influences the cost of access plans, regular collection of statistics increases the likelihood that Trafodion SQL chooses efficient access plans.

While UPDATE STATISTICS is running on a table, the table is active and available for query access.

When a user table is changed, either by changing its data significantly or its definition, re-execute the UPDATE STATISTICS statement for the table.

Histogram Statistics

Histogram statistics are used by the compiler to produce the best plan for a given SQL query. When histograms are not available, default assumptions are made by the compiler and the resultant plan might not perform well. Histograms that reflect the latest data in a table are optimal.

The compiler does not need histogram statistics for every column of a table. For example, if a column is only in the select list, its histogram statistics will be irrelevant. A histogram statistic is useful when a column appears in:

- A predicate
- A GROUP BY column
- An ORDER BY clause
- A HAVING clause
- Or similar clause

In addition to single-column histogram statistics, the compiler needs multi-column histogram statistics, such as when `group by column-5, column-3, column-19` appears in a query. Then, histogram statistics for the combination (`column-5, column-3, column-19`) are needed.

Required Privileges

To perform an UPDATE STATISTICS operation, one of the following must be true:

- You are DB__ROOT.
- You are the owner of the target table.
- You have the MANAGE_STATISTICS component privilege for the SQL_OPERATIONS component.

Locking

UPDATE STATISTICS momentarily locks the definition of the user table during the operation but not the user table itself. The UPDATE STATISTICS statement uses READ UNCOMMITTED isolation level for the user table.

Transactions

Do not start a transaction before executing UPDATE STATISTICS. UPDATE STATISTICS runs multiple transactions of its own, as needed. Starting your own transaction in which UPDATE STATISTICS runs could cause the transaction auto abort time to be exceeded during processing.

Generating and Clearing Statistics for Columns

To generate statistics for particular columns, name each column, or name the first and last columns of a sequence of columns in the table. For example, suppose that a table has consecutive columns CITY, STATE, ZIP. This list gives a few examples of possible options you can specify:

Single-Column Group	Single-Column Group Within Parentheses	Multicolumn Group
`ON CITY, STATE, ZIP`	`ON (CITY),(STATE),(ZIP)`	ON (CITY, STATE) or ON (CITY,STATE,ZIP)
`ON CITY TO ZIP`	`ON (CITY) TO (ZIP)`	
`ON ZIP TO CITY`	`ON (ZIP) TO (CITY)`	
`ON CITY, STATE TO ZIP`	`ON (CITY), (STATE) TO (ZIP)`	
`ON CITY TO STATE, ZIP`	`ON (CITY) TO (STATE), (ZIP)`	

The TO specification is useful when a table has many columns, and you want histograms on a subset of columns. Do not confuse (CITY) TO (ZIP) with (CITY, STATE, ZIP), which refers to a multi-column histogram.

You can clear statistics in any combination of columns you specify, not necessarily with the *`column-group-list`* you used to create statistics. However, those statistics will remain until you clear them.

Column Lists and Access Plans

Generate statistics for columns most often used in data access plans for a table—that is, the primary key, indexes defined on the table, and any other columns frequently referenced in predicates in WHERE or GROUP BY clauses of queries issued on the table. Use the EVERY COLUMN option to generate histograms for every individual column or multicolumns that make up the primary key and indexes.

The EVERY KEY option generates histograms that make up the primary key and indexes.

If you often perform a GROUP BY over specific columns in a table, use multi-column lists in the UPDATE STATISTICS statement (consisting of the columns in the GROUP BY clause) to generate histogram statistics that enable the optimizer to choose a better plan. Similarly, when a query joins two tables by two or more columns, multi-column lists (consisting of the columns being joined) help the optimizer choose a better plan.

Automating Update Statistics

To enable update statistics automation, set the Control Query Default (CQD) attribute, USTAT_AUTOMATION_INTERVAL, in a session where you will run update statistics operations. For example:

```
control query default USTAT_AUTOMATION_INTERVAL '1440';
```

The value of USTAT_AUTOMATION_INTERVAL is intended to be an automation interval (in minutes), but, in Trafodion Release 1.0, this value does not act as a timing interval. Instead, any value greater than zero enables update statistics automation.

After enabling update statistics automation, prepare each of the queries that you want to optimize. For example:

```
prepare s from select...;
```

The PREPARE statement causes the Trafodion SQL compiler to compile and optimize a query without executing it. When preparing queries with update statistic automation enabled, any histograms needed by the optimizer that are not present will cause those columns to be marked as needing histograms.

Next, run this UPDATE STATISTICS statement against each table, using ON NECESSARY COLUMN[S] to generate the needed histograms:

```
update statistics for table table-name on necessary columns sample;
```

Examples of UPDATE STATISTICS

- This example generates four histograms for the columns jobcode, empnum, deptnum, and (empnum, deptnum) for the table EMPLOYEE. Depending on the table's size and data distribution, each histogram should contain ten intervals.

```
UPDATE STATISTICS FOR TABLE employee
  ON (jobcode),(empnum, deptnum)
  GENERATE 10 INTERVALS;

--- SQL operation complete.
```

- This example generates histogram statistics using the ON EVERY COLUMN option for the table DEPT. This statement performs a full scan, and Trafodion SQL determines the default number of intervals.

```
UPDATE STATISTICS FOR TABLE dept
  ON EVERY COLUMN;

--- SQL operation complete.
```

- Suppose that a construction company has an ADDRESS table of potential sites and a DEMOLITION_SITES table that contains some of the columns of the ADDRESS table. The primary key is ZIP. Join these two tables on two of the columns in common:

```
SELECT COUNT(AD.number), AD.street,
       AD.city, AD.zip, AD.state
  FROM address AD, demolition_sites DS
  WHERE AD.zip = DS.zip AND AD.type = DS.type
  GROUP BY AD.street, AD.city, AD.zip, AD.state;
```

  To generate statistics specific to this query, enter these statements:

```
UPDATE STATISTICS FOR TABLE address
  ON (street), (city), (state), (zip, type);

UPDATE STATISTICS FOR TABLE demolition_sites
  ON (zip, type);
```

- This example removes all histograms for table DEMOLITION_SITES:

```
UPDATE STATISTICS FOR TABLE demolition_sites CLEAR;
```

- This example selectively removes the histogram for column STREET in table ADDRESS:

```
UPDATE STATISTICS FOR TABLE address ON street CLEAR;
```

4 SQL Language Elements

Trafodion SQL language elements, which include data types, expressions, functions, identifiers, literals, and predicates, occur within the syntax of SQL statements. The statement and command topics support the syntactical and semantic descriptions of the language elements in this section.

This section describes:

- "Authorization IDs"
- "Character Sets"
- "Columns"
- "Constraints"
- "Correlation Names"
- "Database Objects"
- "Database Object Names"
- "Data Types"
- "Expressions"
- "Identifiers"
- "Indexes"
- "Keys"
- "Literals"
- "Null"
- "Predicates"
- "Roles"
- "Privileges"
- "Schemas"
- "Search Condition"
- "Subquery"
- "Tables"
- "Views"

Authorization IDs

An authorization ID is used for an authorization operation. Authorization is the process of validating that a database user has permission to perform a specified SQL operation. Externally, the authorization ID is a regular or delimited case-insensitive identifier that can have a maximum of 128 characters. See "Case-Insensitive Delimited Identifiers" (page 221). Internally, the authorization ID is associated with a 32-bit number that the database generates and uses for efficient access and storage.

All authorization IDs share the same namespace. An authorization ID can be a database username or a role name. Therefore, a database user and a role cannot share the same name.

An authorization ID can be the PUBLIC authorization ID, which represents all present and future authorization IDs. An authorization ID cannot be _SYSTEM, which is the implicit grantor of privileges to the creator of objects.

Character Sets

You can specify ISO88591 or UTF8 for a character column definition. The use of UTF8 permits you to store characters from many different languages.

Columns

A column is a vertical component of a table and is the relational representation of a field in a record. A column contains one data value for each row of the table.

A column value is the smallest unit of data that can be selected from or updated in a table. Each column has a name that is an SQL identifier and is unique within the table or view that contains the column.

Column References

A qualified column name, or column reference, is a column name qualified by the name of the table or view to which the column belongs, or by a correlation name.

If a query refers to columns that have the same name but belong to different tables, you must use a qualified column name to refer to the columns within the query. You must also refer to a column by a qualified column name if you join a table with itself within a query to compare one row of the table with other rows in the same table.

The syntax of a column reference or qualified column name is:

```
{table-name | view-name | correlation-name}.column-name
```

If you define a correlation name for a table in the FROM clause of a statement, you must use that correlation name if you need to qualify the column name within the statement.

If you do not define an explicit correlation name in the FROM clause, you can qualify the column name with the name of the table or view that contains the column. See "Correlation Names" (page 196).

Derived Column Names

A derived column is an SQL value expression that appears as an item in the select list of a SELECT statement. An explicit name for a derived column is an SQL identifier associated with the derived column. The syntax of a derived column name is:

```
column-expression [[AS] column-name]
```

The column expression can simply be a column reference. The expression is optionally followed by the AS keyword and the name of the derived column.

If you do not assign a name to derived columns, the headings for unnamed columns in query result tables appear as (EXPR). Use the AS clause to assign names that are meaningful to you, which is important if you have more than one derived column in your select list.

Examples of Derived Column Names

These two examples show how to use names for derived columns.

The first example shows (EXPR) as the column heading of the SELECT result table:

```
SELECT AVG (salary)
FROM persnl.employee;
(EXPR)
----------------
        49441.52

--- 1 row(s) selected.
```

The second example shows AVERAGE SALARY as the column heading:

```
SELECT AVG (salary) AS "AVERAGE SALARY"
FROM persnl.employee;
"AVERAGE SALARY"
----------------
        49441.52

--- 1 row(s) selected.
```

Column Default Settings

You can define specific default settings for columns when the table is created. The CREATE TABLE statement defines the default settings for columns within tables. The default setting for a column is the value inserted in a row when an INSERT statement omits a value for a particular column.

Constraints

An SQL constraint is an object that protects the integrity of data in a table by specifying a condition that all the values in a particular column or set of columns of the table must satisfy.

Trafodion SQL enforces these constraints on SQL tables:

CHECK	Column or table constraint specifying a condition must be satisfied for each row in the table.
FOREIGN KEY	Column or table constraint that specifies a referential constraint for the table, declaring that a column or set of columns (called a foreign key) in a table can contain only values that match those in a column or set of columns in the table specified in the REFERENCES clause.
NOT NULL	Column constraint specifying the column cannot contain nulls.
PRIMARY KEY	Column or table constraint specifying the column or set of columns as the primary key for the table.
UNIQUE	Column or table constraint that specifies that the column or set of columns cannot contain more than one occurrence of the same value or set of values.

Creating or Adding Constraints on SQL Tables

To create constraints on an SQL table when you create the table, use the NOT NULL, UNIQUE, CHECK, FOREIGN KEY, or PRIMARY KEY clause of the CREATE TABLE statement.

For more information on Trafodion SQL commands, see "CREATE TABLE Statement" (page 69) and "ALTER TABLE Statement" (page 36).

Constraint Names

When you create a constraint, you can specify a name for it or allow a name to be generated by Trafodion SQL. You can optionally specify both column and table constraint names. Constraint names are ANSI logical names. See "Database Object Names" (page 198). Constraint names are in the same namespace as tables and views, so a constraint name cannot have the same name as a table or view.

The name you specify can be fully qualified or not. If you specify the schema parts of the name, they must match those parts of the affected table and must be unique among table, view, and constraint names in that schema. If you omit the schema portion of the name you specify, Trafodion SQL expands the name by using the schema for the table.

If you do not specify a constraint name, Trafodion SQL constructs an SQL identifier as the name for the constraint and qualifies it with the schema of the table. The identifier consists of the table name concatenated with a system-generated unique identifier.

Correlation Names

A correlation name is a name you can associate with a table reference that is a table, view, or subquery in a SELECT statement to:

- Distinguish a table or view from another table or view referred to in a statement
- Distinguish different uses of the same table
- Make the query shorter

A correlation name can be explicit or implicit.

Explicit Correlation Names

An explicit correlation name for a table reference is an SQL identifier associated with the table reference in the FROM clause of a SELECT statement. See "Identifiers" (page 221). The correlation name must be unique within the FROM clause. For more information about the FROM clause, table references, and correlation names, see "SELECT Statement" (page 138).

The syntax of a correlation name for the different forms of a table reference within a FROM clause is the same:

```
{table | view | (query-expression)} [AS]correlation-name
```

A table or view is optionally followed by the AS keyword and the correlation name. A derived table, resulting from the evaluation of a query expression, must be followed by the AS keyword and the correlation name. An explicit correlation name is known only to the statement in which you define it. You can use the same identifier as a correlation name in another statement.

Implicit Correlation Names

A table or view reference that has no explicit correlation name has an implicit correlation name. The implicit correlation name is the table or view name qualified with the schema names.

You cannot use an implicit correlation name for a reference that has an explicit correlation name within the statement.

Examples of Correlation Names

This query refers to two tables, ORDERS and CUSTOMER, that contain columns named CUSTNUM. In the WHERE clause, one column reference is qualified by an implicit correlation name (ORDERS) and the other by an explicit correlation name (C):

```
SELECT ordernum, custname
FROM orders, customer c
WHERE orders.custnum = c.custnum
  AND orders.custnum = 543;
```

Database Objects

A database object is an SQL entity that exists in a namespace. SQL statements can access Trafodion SQL database objects. The subsections listed below describe these Trafodion SQL database objects.

"Constraints"

"Indexes"

"Tables"

"Views"

Ownership

In Trafodion SQL, the creator of an object owns the object defined in the schema and has all privileges on the object. In addition, you can use the GRANT and REVOKE statements to grant access privileges for a table or view to specified users.

For more information, see the "GRANT Statement" (page 111) and "REVOKE Statement" (page 130). For information on privileges on tables and views, see "CREATE TABLE Statement" (page 69) and "CREATE VIEW Statement" (page 81).

Database Object Names

- "Logical Names for SQL Objects"
- "SQL Object Namespaces"

DML statements can refer to Trafodion SQL database objects. To refer to a database object in a statement, use an appropriate database object name. For information on the types of database objects see "Database Objects" (page 197).

Logical Names for SQL Objects

You may refer to an SQL table, view, constraint, library, function, or procedure by using a one-part, two-part, or three-part logical name, also called an ANSI name:

```
catalog-name.schema-name.object-name
```

In this three-part name, *catalog-name* is the name of the catalog, which is TRAFODION for Trafodion SQL objects that map to HBase tables. *schema-name* is the name of the schema, and *object-name* is the simple name of the table, view, constraint, library, function, or procedure. Each of the parts is an SQL identifier. See "Identifiers" (page 221).

Trafodion SQL automatically qualifies an object name with a schema name unless you explicitly specify schema names with the object name. If you do not set a schema name for the session using a SET SCHEMA statement, the default schema is SEABASE, which exists in the TRAFODION catalog. See "SET SCHEMA Statement" (page 156). A one-part name *object-name* is qualified implicitly with the default schema.

You can qualify a column name in a Trafodion SQL statement by using a three-part, two-part, or one-part object name, or a correlation name.

SQL Object Namespaces

Trafodion SQL objects are organized in a hierarchical manner. Database objects exist in schemas, which are themselves contained in a catalog called TRAFODION. A catalog is a collection of schemas. Schema names must be unique within the catalog.

Multiple objects with the same name can exist provided that each belongs to a different namespace. Trafodion SQL supports these namespaces:

- Index
- Functions and procedures
- Library
- Schema label
- Table value object (table, view, constraint)

Objects in one schema can refer to objects in a different schema. Objects of a given namespace are required to have unique names within a given schema.

Data Types

Trafodion SQL data types are character, datetime, interval, or numeric (exact or approximate):

"Character String Data Types" (page 204)	Fixed-length and variable-length character data types.
"Datetime Data Types" (page 205)	DATE, TIME, and TIMESTAMP data types.
"Interval Data Types" (page 207)	Year-month intervals (years and months) and day-time intervals (days, hours, minutes, seconds, and fractions of a second).
"Numeric Data Types " (page 209)	Exact and approximate numeric data types.

Each column in a table is associated with a data type. You can use the CAST expression to convert data to the data type that you specify. For more information, see "CAST Expression" (page 299).

The following table summarizes the Trafodion SQL data types:

Type	SQL Designation	Description	Size or Range (1)
Fixed-length character	CHAR[ACTER]	Fixed-length character data	1 to 32707 characters (2)
	NCHAR	Fixed-length character data in predefined national character set	1 to 32707 bytes (3) (7)
	NATIONAL CHAR[ACTER]	Fixed-length character data in predefined national character set	1 to 32707 characters (3) (7)
Variable-length character	VARCHAR	Variable-length ASCII character string	1 to 32703 characters (4)
	CHAR[ACTER] VARYING	Variable-length ASCII character string	1 to 32703 characters (4)
	NCHAR VARYING	Variable-length ASCII character string	1 to 32703 bytes (4) (8)
	NATIONAL CHAR[ACTER] VARYING	Variable-length ASCII character string	1 to 32703 characters (4) (8)
Numeric	NUMERIC (1,*scale*) to NUMERIC (128,*scale*)	Binary number with optional scale; signed or unsigned for 1 to 9 digits	1 to 128 digits; stored: 1 to 4 digits in 2 bytes 5 to 9 digits in 4 bytes 10 to 128 digits in 8-64 bytes, depending on precision
	SMALLINT	Binary integer; signed or unsigned	0 to 65535 unsigned, -32768 to +32767 signed; stored in 2 bytes
	INTEGER	Binary integer; signed or unsigned	0 to 4294967295 unsigned, -2147483648 to +2147483647 signed; stored in 4 bytes
	LARGEINT	Binary integer; signed only	-2**63 to +(2**63)-1; stored in 8 bytes
Numeric (extended numeric precision)	NUMERIC (precision 19 to 128)	Binary integer; signed or unsigned	Stored as multiple chunks of 16-bit integers, with a minimum storage length of 8 bytes.
Floating point number	FLOAT[(*precision*)]	Floating point number; precision designates from 1 through 52 bits of precision	+/- 2.2250738585072014e-308 through +/-1.7976931348623157e+308; stored in 8 bytes

Type	SQL Designation	Description	Size or Range (1)
	REAL	Floating point number (32 bits)	+/- 1.17549435e-38 through +/ 3.40282347e+38; stored in 4 bytes
	DOUBLE PRECISION	Floating-point numbers (64 bits) with 1 through 52 bits of precision (52 bits of binary precision and 11 bits of exponent)	+/- 2.2250738585072014e-308 through +/-1.7976931348623157e+308; stored in 8 byte
Decimal number	DECIMAL (1,*scale*) to DECIMAL (18,*scale*)	Decimal number with optional scale; stored as ASCII characters; signed or unsigned for 1 to 9 digits; signed required for 10 or more digits	1 to 18 digits. Byte length equals the number of digits. Sign is stored as the first bit of the leftmost byte.
Date-Time		Point in time, using the Gregorian calendar and a 24 hour clock system. The five supported designations are listed below.	YEAR 0001-9999 MONTH 1-12 DAY 1-31 DAY constrained by MONTH and YEAR HOUR 0-23 MINUTE 0-59 SECOND 0-59 FRACTION(n) 0-999999 in which n is the number of significant digits, from 1 to 6 (default is 6; minimum is 1; maximum is 6). Actual database storage is incremental, as follows: YEAR in 2 bytes MONTH in 1 byte DAY in 1 byte HOUR in 1 byte MINUTE in 1 byte SECOND in 1 byte FRACTION in 4 bytes
	DATE	Date	Format as YYYY-MM-DD; actual database storage size is 4 bytes
	TIME	Time of day, 24 hour clock, no time precision	Format as HH:MM:SS; actual database storage size is 3 bytes
	TIME (with time precision)	Time of day, 24 hour clock, with time precision	Format as HH:MM:SS.FFFFFF; actual database storage size is 7 bytes
	TIMESTAMP	Point in time, no time precision	Format as YYYY-MM-DD HH:MM:SS; actual database storage size is 7 bytes
	TIMESTAMP (with time precision)	Point in time, with time precision	Format as YYYY-MM-DD HH:MM:SS.FFFFFF; actual database storage size is 11 bytes
Interval	INTERVAL	Duration of time; value is in the YEAR/MONTH range or the DAY/HOUR/MINUTE/ SECOND/FRACTION range	YEAR no constraint(6) MONTH 0-11 DAY no constraint

Type	SQL Designation	Description	Size or Range (1)
			HOUR 0-23 MINUTE 0-59 SECOND 0-59 FRACTION(n) 0-999999 in which n is the number of significant digits (default is 6; minimum is 1; maximum is 6); stored in 2, 4, or 8 bytes depending on number of digits

scale is the number of digits to the right of the decimal.

precision specifies the allowed number of decimal digits.

(1) The size of a column that allows null values is 2 bytes larger than the size for the defined data type.

(2) The maximum row size is 32708 bytes, but the actual row size is less than that because of bytes used by null indicators, varchar column length indicators, and actual data encoding.

(3) Storage size is the same as that required by CHAR data type but store only half as many characters depending on character set selection.

(4) Storage size is reduced by 4 bytes for storage of the varying character length.

(5) The maximum number of digits in an INTERVAL value is 18, including the digits in all INTERVAL fields of the value. Any INTERVAL field that is a starting field can have up to 18 digits minus the number of other digits in the INTERVAL value.

(6) The maximum is 32707 if the national character set was specified at installation time to be ISO88591. The maximum is 16353 if the national character set was specified at installation time as UTF8.

(7) The maximum is 32703 if the national character set was specified at installation time to be ISO88591. The maximum is 16351 if the national character set was specified at installation time as UTF8.

Comparable and Compatible Data Types

Two data types are comparable if a value of one data type can be compared to a value of the other data type.

Two data types are compatible if a value of one data type can be assigned to a column of the other data type, and if columns of the two data types can be combined using arithmetic operations. Compatible data types are also comparable.

Assignment and comparison are the basic operations of Trafodion SQL. Assignment operations are performed during the execution of INSERT and UPDATE statements. Comparison operations are performed during the execution of statements that include predicates, aggregate (or set) functions, and GROUP BY, HAVING, and ORDER BY clauses.

The basic rule for both assignment and comparison is that the operands have compatible data types. Data types with different character sets cannot be compared without converting one character set to the other. However, the SQL compiler will usually generate the necessary code to do this conversion automatically.

Character Data Types

Values of fixed and variable length character data types of the same character set are all character strings and are all mutually comparable and mutually assignable.

When two strings are compared, the comparison is made with a temporary copy of the shorter string that has been padded on the right with blanks to have the same length as the longer string.

Datetime Data Types

Values of type datetime are mutually comparable and mutually assignable only if the types have the same datetime fields. A DATE, TIME, or TIMESTAMP value can be compared with another value only if the other value has the same data type.

All comparisons are chronological. For example, this predicate is true:

```
TIMESTAMP '2008-09-28 00:00:00' >
  TIMESTAMP '2008-06-26 00:00:00'
```

Interval Data Types

Values of type INTERVAL are mutually comparable and mutually assignable only if the types are either both year-month intervals or both day-time intervals.

For example, this predicate is true:

```
INTERVAL '02-01' YEAR TO MONTH > INTERVAL '00-01' YEAR TO MONTH
```

The field components of the INTERVAL do not have to be the same. For example, this predicate is also true:

```
INTERVAL '02-01' YEAR TO MONTH > INTERVAL '01' YEAR
```

Numeric Data Types

Values of the approximate data types FLOAT, REAL, and DOUBLE PRECISION, and values of the exact data types NUMERIC, DECIMAL, INTEGER, SMALLINT, and LARGEINT, are all numbers and are all mutually comparable and mutually assignable.

When an approximate data type value is assigned to a column with exact data type, rounding might occur, and the fractional part might be truncated. When an exact data type value is assigned to a column with approximate data type, the result might not be identical to the original number.

When two numbers are compared, the comparison is made with a temporary copy of one of the numbers, according to defined rules of conversion. For example, if one number is INTEGER and the other is DECIMAL, the comparison is made with a temporary copy of the integer converted to a decimal.

Extended Numeric Precision

Trafodion SQL provides support for extended numeric precision data type. Extended numeric precision is an extension to the NUMERIC(x,y) data type where no theoretical limit exists on precision. It is a software data type, which means that the underlying hardware does not support it and all computations are performed by software. Computations using this data type may not match the performance of other hardware supported data types.

Considerations for Extended NUMERIC Precision Data Type

Consider these points and limitations for extended NUMERIC precision data type:

- May cost more than other data type options.
- Is a software data type.
- Cannot be compared to data types that are supported by hardware.
- If your application requires extended NUMERIC precision arithmetic expressions, specify the required precision in the table DDL or as explicit extended precision type casts of your select list items. The default system behavior is to treat user-specified extended precision expressions as extended precision values. Conversely, non-user-specified (that is, temporary, intermediate) extended precision expressions may lose precision. In the following example, the precision appears to lose one digit because the system treats the sum of two NUMERIC(18,4) type columns as NUMERIC(18,4). NUMERIC(18) is the longest non-extended precision numeric type. NUMERIC(19) is the shortest extended precision numeric type. The system actually computes the sum of 2 NUMERIC(18,4) columns as an extended precision NUMERIC(19,4) sum. But because no user-specified extended precision columns exist, the system casts the sum back to the user-specified type of NUMERIC(18,4).

  ```
  CREATE TABLE T(a NUMERIC(18,4), B NUMERIC(18,4));

  INSERT INTO T VALUES (1.1234, 2.1234);
  ```

```
>> SELECT A+B FROM T;

(EXPR)
--------------
         3.246
```

If this behavior is not acceptable, you can use one of these options:

- Specify the column type as NUMERIC(19,4). For example, `CREATE TABLE T(A NUMERIC(19,4), B NUMERIC(19,4));` or
- Cast the sum as NUMERIC(19,4). For example, `SELECT CAST(A+B AS NUMERIC(19,4)) FROM T;` or
- Use an extended precision literal in the expression. For example, `SELECT A+B*1.00000000000000000000 FROM T;`.

Note the result for the previous example when changing to NUMERIC(19,4):

```
SELECT CAST(A+B AS NUMERIC(19,4)) FROM T;

(EXPR)
------------
      3.2468
```

When displaying output results in the command interface of a client-based tool, casting a select list item to an extended precision numeric type is acceptable. However, when retrieving an extended precision select list item into an application program's host variable, you must first convert the extended precision numeric type into a string data type. For example:

```
SELECT CAST(CAST(A+B AS NUMERIC(19,4)) AS CHAR(24)) FROM T;

(EXPR)
------------
      3.2468
```

NOTE: An application program can convert an externalized extended precision value in string form into a numeric value it can handle. But, an application program cannot correctly interpret an extended precision value in internal form.

Rules for Extended NUMERIC Precision Data Type

These rules apply:

- No limit on maximum precision.
- Supported in all DDL and DML statements where regular NUMERIC data type is supported.
- Allowed as part of key columns for hash partitioned tables only.
- NUMERIC type with precision 10 through 18.
  - UNSIGNED is supported as extended NUMERIC precision data type
  - SIGNED is supported as 64-bit integer
- CAST function allows conversion between regular NUMERIC and extended NUMERIC precision data type.
- Parameters in SQL queries support extended NUMERIC precision data type.

Example of Extended NUMERIC Precision Data Type

```
>>CREATE TABLE t( n NUMERIC(128,30));

--- SQL operation complete.
```

```
>>SHOWDDL TABLE t;

CREATE TABLE SCH.T
  (
    N           NUMERIC(128, 30) DEFAULT NULL
  )
  ;

--- SQL operation complete.
>>
```

Character String Data Types

Trafodion SQL includes both fixed-length character data and variable-length character data. You cannot compare character data to numeric, datetime, or interval data.

```
character-type is:
     CHAR[ACTER] [(length [CHARACTERS])] [char-set] [UPSHIFT] [[NOT]CASESPECIFIC]
   | CHAR[ACTER] VARYING(length) [CHARACTERS][char-set] [UPSHIFT] [[NOT]CASESPECIFIC]
   | VARCHAR(length) [CHARACTERS] [char-set] [UPSHIFT] [[NOT]CASESPECIFIC]
   | NCHAR [(length)] [CHARACTERS] [UPSHIFT] [[NOT]CASESPECIFIC]
   | NCHAR VARYING (length) [CHARACTERS] [UPSHIFT] [[NOT]CASESPECIFIC]
   | NATIONAL CHAR[ACTER] [(length)] [CHARACTERS] [UPSHIFT] [[NOT]CASESPECIFIC]
   | NATIONAL CHAR[ACTER] VARYING (length) [CHARACTERS] [UPSHIFT] [[NOT]CASESPECIFIC]

char-set is
  CHARACTER SET char-set-name
```

CHAR, NCHAR, and NATIONAL CHAR are fixed-length character types. CHAR VARYING, VARCHAR, NCHAR VARYING and NATIONAL CHAR VARYING are varying-length character types.

length

is a positive integer that specifies the number of characters allowed in the column. You must specify a value for *length*.

char-set-name

is the character set name, which can be ISO88591 or UTF8.

CHAR[ACTER] [(*length* [CHARACTERS])] [*char-set*] [UPSHIFT] [[NOT]CASESPECIFIC]

specifies a column with fixed-length character data.

CHAR[ACTER] VARYING (*length*) [CHARACTERS] [*char-set*] [UPSHIFT] [[NOT]CASESPECIFIC]

specifies a column with varying-length character data. VARYING specifies that the number of characters stored in the column can be fewer than the *length*.

Values in a column declared as VARYING can be logically and physically shorter than the maximum length, but the maximum internal size of a VARYING column is actually four bytes larger than the size required for an equivalent column that is not VARYING.

VARCHAR (*length*) [*char-set*] [UPSHIFT] [[NOT]CASESPECIFIC]

specifies a column with varying-length character data.

VARCHAR is equivalent to data type CHAR[ACTER] VARYING.

NCHAR [(*length*)] [UPSHIFT] [[NOT]CASESPECIFIC], NATIONAL CHAR[ACTER] [(*length*)] [UPSHIFT] [[NOT]CASESPECIFIC]

specifies a column with data in the predefined national character set.

NCHAR VARYING [(*length*)] [UPSHIFT] [[NOT]CASESPECIFIC], NATIONAL CHAR[ACTER] VARYING (*length*) [UPSHIFT] [[NOT]CASESPECIFIC]

specifies a column with varying-length data in the predefined national character set.

Considerations for Character String Data Types

Difference Between CHAR and VARCHAR

You can specify a fixed-length character column as CHAR(*n*), where *n* is the number of characters you want to store. However, if you store five characters into a column specified as CHAR(10), ten characters are stored where the rightmost five characters are blank.

If you do not want to have blanks added to your character string, you can specify a variable-length character column as VARCHAR(*n*), where *n* is the maximum number of characters you want to store. If you store five characters in a column specified as VARCHAR(10), only the five characters are stored logically—without blank padding.

NCHAR Columns in SQL Tables

In Trafodion SQL, the NCHAR type specification is equivalent to:

- NATIONAL CHARACTER
- NATIONAL CHAR
- CHAR ... CHARACTER SET ..., where the character set is the character set for NCHAR

Similarly, you can use NCHAR VARYING, NATIONAL CHARACTER VARYING, NATIONAL CHAR VARYING, and VARCHAR ... CHARACTER SET ... , where the character set is the character set for NCHAR. The character set for NCHAR is determined when Trafodion SQL is installed.

Datetime Data Types

A value of datetime data type represents a point in time according to the Gregorian calendar and a 24-hour clock in local civil time (LCT). A datetime item can represent a date, a time, or a date and time.

When a numeric value is added to or subtracted from a date type, the numeric value is automatically CASTed to an INTERVAL DAY value. When a numeric value is added to or subtracted from a time type or a timestamp type, the numeric value is automatically CASTed to an INTERVAL SECOND value. For information on CAST, see "CAST Expression" (page 299).

Trafodion SQL accepts dates, such as October 5 to 14, 1582, that were omitted from the Gregorian calendar. This functionality is a Trafodion SQL extension.

The range of times that a datetime value can represent is:

January 1, 1 A.D., 00:00:00.000000 (low value) December 31, 9999, 23:59:59.999999 (high value)

Trafodion SQL has three datetime data types:

```
datetime-type is:
     DATE
   | TIME [(time-precision)]
   | TIMESTAMP [(timestamp-precision)]
```

DATE

specifies a datetime column that contains a date in the external form yyyy-mm-dd and stored in four bytes.

TIME [(*time-precision*)]

specifies a datetime column that, without the optional time-precision, contains a time in the external form hh:mm:ss and is stored in three bytes. *time-precision* is an unsigned integer that specifies the number of digits in the fractional seconds and is stored in four bytes. The default for *time-precision* is 0, and the maximum is 6.

TIMESTAMP [(*timestamp-precision*)]

specifies a datetime column that, without the optional *timestamp-precision*, contains a timestamp in the external form yyyy-mm-dd hh:mm:ss and is stored in seven bytes. *timestamp-precision* is an unsigned integer that specifies the number of digits in the

fractional seconds and is stored in four bytes. The default for *timestamp-precision* is 6, and the maximum is 6.

Considerations for Datetime Data Types

Datetime Ranges

The range of values for the individual fields in a DATE, TIME, or TIMESTAMP column is specified as:

yyyy	Year, from 0001 to 9999
mm	Month, from 01 to 12
dd	Day, from 01 to 31
hh	Hour, from 00 to 23
mm	Minute, from 00 to 59
ss	Second, from 00 to 59
mssssss	Microsecond, from 000000 to 999999

When you specify *datetime_value* (FORMAT 'string') in the DML statement and the specified format is 'mm/dd/yyyy','MM/DD/YYYY', or 'yyyy/mm/dd' or 'yyyy-mm-dd', the datetime type is automatically cast.

Interval Data Types

Values of interval data type represent durations of time in year-month units (years and months) or in day-time units (days, hours, minutes, seconds, and fractions of a second).

```
interval-type is:
INTERVAL[-] { start-field TO end-field | single-field }

start-field is:
   {YEAR | MONTH | DAY | HOUR | MINUTE} [(leading-precision)]

end-field is:
   YEAR | MONTH | DAY | HOUR | MINUTE | SECOND
        [(fractional-precision)]

single-field is:
   start-field | SECOND [(leading-precision,
                          fractional-precision)]
```

INTERVAL[-] { *start-field* TO *end-field* | *single-field* }

specifies a column that represents a duration of time as a year-month or day-time range or a single-field. The optional sign indicates if this is a positive or negative integer. If you omit the sign, it defaults to positive.

If the interval is specified as a range, the *start-field* and *end-field* must be in one of these categories:

{YEAR | MONTH | DAY | HOUR | MINUTE} [(*leading-precision*)]

specifies the *start-field*. A *start-field* can have a *leading-precision* up to 18 digits (the maximum depends on the number of fields in the interval). The *leading-precision* is the number of digits allowed in the *start-field*. The default for *leading-precision* is 2.

YEAR | MONTH | DAY | HOUR | MINUTE | SECOND [(*fractional-precision*)]

specifies the *end-field*. If the *end-field* is SECOND, it can have a *fractional-precision* up to 6 digits. The *fractional-precision* is the number of digits of precision after the decimal point. The default for *fractional-precision* is 6.

start-field | SECOND [(*leading-precision*, *fractional-precision*)]

specifies the *single-field*. If the *single-field* is SECOND, the *leading-precision* is the number of digits of precision before the decimal point, and the *fractional-precision* is the number of digits of precision after the decimal point.

The default for *leading-precision* is 2, and the default for *fractional-precision* is 6. The maximum for *leading-precision* is 18, and the maximum for *fractional-precision* is 6.

Considerations for Interval Data Types

Adding or Subtracting Imprecise Interval Values

Adding or subtracting an interval that is any multiple of a MONTH, a YEAR, or a combination of these may result in a runtime error. For example, adding 1 MONTH to January 31, 2009 will result in an error because February 31 does not exist and it is not clear whether the user would want rounding back to February 28, 2009, rounding up to March 1, 2009 or perhaps treating the interval 1 MONTH as if it were 30 days resulting in an answer of March 2, 2009. Similarly, subtracting 1 YEAR from February 29, 2008 will result in an error. See the descriptions for the "ADD_MONTHS Function" (page 287), "DATE_ADD Function" (page 320), "DATE_SUB Function" (page 321), and "DATEADD Function" (page 322) for ways to add or subtract such intervals without getting errors at runtime.

Interval Leading Precision

The maximum for the *leading-precision* depends on the number of fields in the interval and on the *fractional-precision*. The maximum is computed as:

```
max-leading-precision = 18 - fractional-precision - 2 * (N - 1)
```

where *N* is the number of fields in the interval.

For example, the maximum number of digits for the *leading-precision* in a column with data type INTERVAL YEAR TO MONTH is computed as: 18 – 0 – 2 * (2 – 1) = 16

Interval Ranges

Within the definition of an interval range (other than a single field), the *start-field* and *end-field* can be any of the specified fields with these restrictions:

- An interval range is either year-month or day-time—that is, if the *start-field* is YEAR, the *end-field* is MONTH; if the *start-field* is DAY, HOUR, or MINUTE, the *end-field* is also a time field.
- The *start-field* must precede the *end-field* within the hierarchy: YEAR, MONTH, DAY, HOUR, MINUTE, and SECOND.

Signed Intervals

To include a quoted string in a signed interval data type, the sign must be outside the quoted string. It can be before the entire literal or immediately before the duration enclosed in quotes.

For example, for the interval "minus (5 years 5 months) these formats are valid:

```
INTERVAL - '05-05'YEAR TO MONTH
- INTERVAL '05-05' YEAR TO MONTH
```

Overflow Conditions

When you insert a fractional value into an INTERVAL data type field, if the fractional value is 0 (zero) it does not cause an overflow. Inserting value INTERVAL '1.000000' SECOND(6) into a field SECOND(0) does not cause a loss of value. Provided that the value fits in the target column without a loss of precision, Trafodion SQL does not return an overflow error.

However, if the fractional value is > 0, an overflow occurs. Inserting value INTERVAL '1.000001' SECOND(6) causes a loss of value.

Numeric Data Types

Numeric data types are either exact or approximate. A numeric data type is compatible with any other numeric data type, but not with character, datetime, or interval data types.

```
exact-numeric-type is:
     NUMERIC [(precision [,scale])] [SIGNED|UNSIGNED]
   | SMALLINT [SIGNED|UNSIGNED]
   | INT[EGER] [SIGNED|UNSIGNED]
   | LARGEINT
   | DEC[IMAL] [(precision [,scale])] [SIGNED|UNSIGNED]
approximate-numeric-type is:
     FLOAT [(precision)]
   | REAL
   | DOUBLE PRECISION
```

Exact numeric data types are types that can represent a value exactly: NUMERIC, SMALLINT, INTEGER, LARGEINT, and DECIMAL.

Approximate numeric data types are types that do not necessarily represent a value exactly: FLOAT, REAL, and DOUBLE PRECISION.

A column in a Trafodion SQL table declared with a floating-point data type is stored in IEEE floating-point format and all computations on it are done assuming that. Trafodion SQL tables can contain only IEEE floating-point data.

`NUMERIC [(`*`precision`* `[,`*`scale`*`])] [SIGNED|UNSIGNED]`

specifies an exact numeric column—a two-byte binary number, `SIGNED` or `UNSIGNED`. *`precision`* specifies the total number of digits and cannot exceed 128. If *`precision`* is between 10 and 18, you must use a signed value to obtain the supported hardware data type. If precision is over 18, you will receive the supported software data type. You will also receive the supported software data type if the precision type is between 10 and 18, and you specify UNSIGNED. *`scale`* specifies the number of digits to the right of the decimal point.

The default is `NUMERIC (9,0) SIGNED`.

`SMALLINT [SIGNED|UNSIGNED]`

specifies an exact numeric column—a two-byte binary integer, `SIGNED` or `UNSIGNED`. The column stores integers in the range unsigned 0 to 65535 or signed -32768 to +32767.

The default is `SIGNED`.

`INT[EGER] [SIGNED|UNSIGNED]`

specifies an exact numeric column—a 4-byte binary integer, `SIGNED` or `UNSIGNED`. The column stores integers in the range unsigned 0 to 4294967295 or signed -2147483648 to +2147483647.

The default is `SIGNED`.

`LARGEINT`

specifies an exact numeric column—an 8-byte signed binary integer. The column stores integers in the range -2**63 to +2**63 -1 (approximately 9.223 times 10 to the eighteenth power).

`DEC[IMAL] [(`*`precision`* `[,`*`scale`*`])] [SIGNED|UNSIGNED]`

specifies an exact numeric column—a decimal number, `SIGNED` or `UNSIGNED`,stored as ASCII characters. *`precision`* specifies the total number of digits and cannot exceed 18. If *`precision`* is 10 or more, the value must be `SIGNED`. The sign is stored as the first bit of the leftmost byte. *`scale`* specifies the number of digits to the right of the decimal point.

The default is `DECIMAL (9,0) SIGNED`.

`FLOAT [( precision )]`

specifies an approximate numeric column. The column stores floating-point numbers and designates from 1 through 54 bits of *`precision`*. The range is from +/- 2.2250738585072014e-308 through +/-1.7976931348623157e+308 stored in 8 bytes.

An IEEE FLOAT *precision* data type is stored as an IEEE DOUBLE, that is, in 8 bytes, with the specified precision.

The default *precision* is 54.

REAL

specifies a 4-byte approximate numeric column. The column stores 32-bit floating-point numbers with 23 bits of binary precision and 8 bits of exponent.

The minimum and maximum range is from +/- 1.17549435e-38 through +/ 3.40282347e+38.

DOUBLE PRECISION

specifies an 8-byte approximate numeric column.

The column stores 64-bit floating-point numbers and designates from 1 through 52 bits of *precision*.

An IEEE DOUBLE PRECISION data type is stored in 8 bytes with 52 bits of binary precision and 11 bits of exponent. The minimum and maximum range is from +/- 2.2250738585072014e-308 through +/-1.7976931348623157e+308.

Expressions

An SQL value expression, called an expression, evaluates to a value. Trafodion SQL supports these types of expressions:

"Character Value Expressions" (page 211)	Operands can be combined with the concatenation operator (||). Example: `'HOUSTON,' ||' TEXAS'`
"Datetime Value Expressions" (page 212)	Operands can be combined in specific ways with arithmetic operators. Example: `CURRENT_DATE + INTERVAL '1' DAY`
"Interval Value Expressions" (page 215)	Operands can be combined in specific ways with addition and subtraction operators. Example: `INTERVAL '2' YEAR - INTERVAL '3' MONTH`
"Numeric Value Expressions" (page 218)	Operands can be combined in specific ways with arithmetic operators. Example: `SALARY * 1.10`

The data type of an expression is the data type of the value of the expression.

A value expression can be a character string literal, a numeric literal, a dynamic parameter, or a column name that specifies the value of the column in a row of a table. A value expression can also include functions and scalar subqueries.

Character Value Expressions

The operands of a character value expression—called character primaries—can be combined with the concatenation operator (||). The data type of a character primary is character string.

```
character-expression is:
     character-primary
   | character-expression || character-primary

character-primary is:
     character-string-literal
   | column-reference
   | character-type-host-variable
   | dynamic parameter
   | character-value-function
   | aggregate-function
   | sequence-function
   | scalar-subquery
   | CASE-expression
   | CAST-expression
   | (character-expression)
```

Character (or string) value expressions are built from operands that can be:

- Character string literals
- Character string functions
- Column references with character values
- Dynamic parameters
- Aggregate functions, sequence functions, scalar subqueries, CASE expressions, or CAST expressions that return character values

Examples of Character Value Expressions

These are examples of character value expressions:

Expression	Description
`'ABILENE'`	Character string literal.
`'ABILENE ' ||' TEXAS'`	The concatenation of two string literals.
`'ABILENE ' ||' TEXAS ' || x'55 53 41'`	The concatenation of three string literals to form the literal: 'ABILENE TEXAS USA'
`'Customer ' || custname`	The concatenation of a string literal with the value in column CUSTNAME.
`CAST (order_date AS CHAR(10))`	CAST function applied to a DATE value.

Datetime Value Expressions

- "Considerations for Datetime Value Expressions"
- "Examples of Datetime Value Expressions"

The operands of a datetime value expression can be combined in specific ways with arithmetic operators.

In this syntax diagram, the data type of a datetime primary is DATE, TIME, or TIMESTAMP. The data type of an interval term is INTERVAL.

```
datetime-expression is:
     datetime-primary
   | interval-expression + datetime-primary
   | datetime-expression + interval-term
   | datetime-expression - interval-term

datetime-primary is:
     datetime-literal
   | column-reference
   | datetime-type-host-variable
   | dynamic parameter
   | datetime-value-function
   | aggregate-function
   | sequence-function
   | scalar-subquery
   | CASE-expression
   | CAST-expression
   | (datetime-expression)

interval-term is:
     interval-factor
   | numeric-term  * interval-factor

interval-factor is:
   [+|-] interval-primary

interval-primary is:
     interval-literal
   | column-reference
   | interval-type-host-variable
   | dynamic parameter
   | aggregate-function
   | sequence-function
   | scalar-subquery
   | CASE-expression
   | CAST-expression
   | (interval-expression)
```

Datetime value expressions are built from operands that can be:

- Interval value expressions
- Datetime or interval literals
- Dynamic parameters
- Column references with datetime or interval values
- Dynamic parameters
- Datetime or interval value functions
- Any aggregate functions, sequence functions, scalar subqueries, CASE expressions, or CAST expressions that return datetime or interval values

Considerations for Datetime Value Expressions

Data Type of Result

In general, the data type of the result is the data type of the *datetime-primary* part of the datetime expression. For example, datetime value expressions include:

Datetime Expression	Description	Result Data Type
`CURRENT_DATE + INTERVAL '1' DAY`	The sum of the current date and an interval value of one day.	DATE
`CURRENT_DATE + est_complete`	The sum of the current date and the interval value in column EST_COMPLETE.	DATE
`( SELECT ship_timestamp FROM project WHERE projcode=1000 ) + INTERVAL '07:04' DAY TO HOUR`	The sum of the ship timestamp for the specified project and an interval value of seven days, four hours.	TIMESTAMP

The datetime primary in the first expression is CURRENT_DATE, a function that returns a value with DATE data type. Therefore, the data type of the result is DATE.

In the last expression, the datetime primary is this scalar subquery:

```
( SELECT ship_timestamp FROM project WHERE projcode=1000 )
```

The preceding subquery returns a value with TIMESTAMP data type. Therefore, the data type of the result is TIMESTAMP.

Restrictions on Operations With Datetime or Interval Operands

You can use datetime and interval operands with arithmetic operators in a datetime value expression only in these combinations:

Operand 1	Operator	Operand 2	Result Type
Datetime	+ or –	Interval	Datetime
Interval	+	Datetime	Datetime

When a numeric value is added to or subtracted from a DATE type, the numeric value is automatically CASTed to an INTERVAL DAY value. When a numeric value is added to or subtracted from a time type or a timestamp type, the numeric value is automatically CASTed to an INTERVAL SECOND value. For information on CAST, see "CAST Expression" (page 299). For more information on INTERVALS, see "Interval Value Expressions" (page 215)

When using these operations, note:

- Adding or subtracting an interval of months to a DATE value results in a value of the same day plus or minus the specified number of months. Because different months have different lengths, this is an approximate result.
- Datetime and interval arithmetic can yield unexpected results, depending on how the fields are used. For example, execution of this expression (evaluated left to right) returns an error:

  ```
  DATE '2007-01-30' + INTERVAL '1' MONTH + INTERVAL '7' DAY
  ```

  In contrast, this expression (which adds the same values as the previous expression, but in a different order) correctly generates the value 2007-03-06:

  ```
  DATE '2007-01-30' + INTERVAL '7' DAY + INTERVAL '1' MONTH
  ```

You can avoid these unexpected results by using the "ADD_MONTHS Function" (page 287).

Examples of Datetime Value Expressions

The PROJECT table consists of five columns that use the data types NUMERIC, VARCHAR, DATE, TIMESTAMP, and INTERVAL DAY. Suppose that you have inserted values into the PROJECT table. For example:

```
INSERT INTO persnl.project
VALUES (1000,'SALT LAKE CITY',DATE '2007-04-10',
       TIMESTAMP '2007-04-21:08:15:00.00',INTERVAL '15' DAY);
```

The next examples use these values in the PROJECT table:

PROJCODE	START_DATE	SHIP_TIMESTAMP	EST_COMPLETE
1000	2007-04-10	2007-04-21 08:15:00.00	15
945	2007-10-20	2007-12-21 08:15:00.00	30
920	2007-02-21	2007-03-12 09:45:00.00	20
134	2007-11-20	2008-01-01 00:00:00.00	30

- Add an interval value qualified by YEAR to a datetime value:

  ```
  SELECT start_date + INTERVAL '1' YEAR
  FROM persnl.project
  WHERE projcode = 1000;

  (EXPR)
  ----------
  2008-04-10

  --- 1 row(s) selected.
  ```

- Subtract an interval value qualified by MONTH from a datetime value:

  ```
  SELECT ship_timestamp - INTERVAL '1' MONTH
  FROM persnl.project
  WHERE projcode = 134;

  (EXPR)
  --------------------------
  2007-12-01 00:00:00.000000

  --- 1 row(s) selected.
  ```

  The result is 2007-12-01 00:00:00.00. The YEAR value is decremented by 1 because subtracting a month from January 1 causes the date to be in the previous year.

- Add a column whose value is an interval qualified by DAY to a datetime value:

  ```
  SELECT start_date + est_complete
  FROM persnl.project
  ```

```
WHERE projcode = 920;

(EXPR)
----------
2007-03-12

--- 1 row(s) selected.
```

The result of adding 20 days to 2008-02-21 is 2008-03-12. Trafodion SQL correctly handles 2008 as a leap year.

- Subtract an interval value qualified by HOUR TO MINUTE from a datetime value:

```
SELECT ship_timestamp - INTERVAL '15:30' HOUR TO MINUTE
FROM persnl.project
WHERE projcode = 1000;

(EXPR)
--------------------------
2008-04-20 16:45:00.000000
```

The result of subtracting 15 hours and 30 minutes from 2007-04-21 08:15:00.00 is 2007-04-20 16:45:00.00.

Interval Value Expressions

- "Considerations for Interval Value Expressions"
- "Examples of Interval Value Expressions"

The operands of an interval value expression can be combined in specific ways with addition and subtraction operators. In this syntax diagram, the data type of a datetime expression is DATE, TIME, or TIMESTAMP; the data type of an interval term or expression is INTERVAL.

```
interval-expression is:
     interval-term
   | interval-expression + interval-term
   | interval-expression - interval-term
   | (datetime-expression - datetime-primary)
        [interval-qualifier]

interval-term is:
     interval-factor
   | interval-term * numeric-factor
   | interval-term / numeric-factor
   | numeric-term  * interval-factor

interval-factor is:
   [+|-] interval-primary

interval-primary is:
     interval-literal
   | column-reference
   | interval-type-host-variable
   | dynamic-parameter
   | aggregate-function
   | sequence-function
   | scalar-subquery
   | CASE-expression
   | CAST-expression
   | (interval-expression)

numeric-factor is:
     [+|-] numeric-primary
   | [+|-] numeric-primary ** numeric-factor
```

Interval value expressions are built from operands that can be:

- Integers
- Datetime value expressions
- Interval literals
- Column references with datetime or interval values
- Dynamic parameters
- Datetime or interval value functions
- Aggregate functions, sequence functions, scalar subqueries, CASE expressions, or CAST expressions that return interval values

For *interval-term*, *datetime-expression*, and *datetime-primary*, see "Datetime Value Expressions" (page 212).

If the interval expression is the difference of two datetime expressions, by default, the result is expressed in the least significant unit of measure for that interval. For date differences, the interval is expressed in days. For timestamp differences, the interval is expressed in fractional seconds.

If the interval expression is the difference or sum of interval operands, the interval qualifiers of the operands are either year-month or day-time. If you are updating or inserting a value that is the result of adding or subtracting two interval qualifiers, the interval qualifier of the result depends on the interval qualifier of the target column.

Considerations for Interval Value Expressions

Start and End Fields

Within the definition of an interval range, the *start-field* and *end-field* can be any of the specified fields with these restrictions:

- An interval is either year-month or day-time. If the *start-field* is YEAR, the *end-field* is MONTH; if the *start-field* is DAY, HOUR, or MINUTE, the *end-field* is also a time field.
- The *start-field* must precede the *end-field* within the hierarchy YEAR, MONTH, DAY, HOUR, MINUTE, and SECOND.

Within the definition of an interval expression, the *start-field* and *end-field* of all operands in the expression must be either year-month or day-time.

Interval Qualifier

The rules for determining the interval qualifier of the result expression vary. For example, interval value expressions include:

Datetime Expression	Description	Result Data Type
`CURRENT_DATE - start_date`	By default, the interval difference between the current date and the value in column START_DATE is expressed in days. You are not required to specify the interval qualifier.	INTERVAL DAY (12)
`INTERVAL '3' DAY - INTERVAL '2' DAY`	The difference of two interval literals. The result is 1 day.	INTERVAL DAY (3)
`INTERVAL '3' DAY + INTERVAL '2' DAY`	The sum of two interval literals. The result is 5 days.	INTERVAL DAY (3)
`INTERVAL '2' YEAR - INTERVAL '3' MONTH`	The difference of two interval literals. The result is 1 year, 9 months.	INTERVAL YEAR (3) TO MONTH

Restrictions on Operations

You can use datetime and interval operands with arithmetic operators in an interval value expression only in these combinations:

Operand 1	Operator	Operand 2	Result Type
Datetime	–	Datetime	Interval
Interval	+ or –	Interval	Interval
Interval	* or /	Numeric	Interval
Numeric	*	Interval	Interval

This table lists valid combinations of datetime and interval arithmetic operators, and the data type of the result:

Operands	Result type
Date + Interval or Interval + Date	Date
Date + Numeric or Numeric + Date	Date
Date - Numeric	Date
Date – Interval	Date
Date – Date	Interval
Time + Interval or Interval + Time	Time
Time + Numeric or Numeric + Time	Time
Time - Number	Time
Time – Interval	Time
Timestamp + Interval or Interval + Timestamp	Timestamp
Timestamp + Numeric or Numeric + Timestamp	Timestamp
Timestamp - Numeric	Timestamp
Timestamp – Interval	Timestamp
year-month Interval + year-month Interval	year-month Interval
day-time Interval + day-time Interval	day-time Interval
year-month Interval – year-month Interval	year-month Interval
day-time Interval – day-time Interval	day-time Interval
Time – Time	Interval
Timestamp – Timestamp	Interval
Interval * Number or Number * Interval	Interval
Interval / Number	Interval
Interval – Interval or Interval + Interval	Interval

When using these operations, note:

- If you subtract a datetime value from another datetime value, both values must have the same data type. To get this result, use the CAST expression. For example:

  ```
  CAST (ship_timestamp AS DATE) - start_date
  ```

- If you subtract a datetime value from another datetime value, and you specify the interval qualifier, you must allow for the maximum number of digits in the result for the precision. For example:

  ```
  (CURRENT_TIMESTAMP - ship_timestamp) DAY(4) TO SECOND(6)
  ```

- If you are updating a value that is the result of adding or subtracting two interval values, an SQL error occurs if the source value does not fit into the target column's range of interval fields. For example, this expression cannot replace an INTERVAL DAY column:

  ```
  INTERVAL '1' MONTH + INTERVAL '7' DAY
  ```

- If you multiply or divide an interval value by a numeric value expression, Trafodion SQL converts the interval value to its least significant subfield and then multiplies or divides it by the numeric value expression. The result has the same fields as the interval that was multiplied or divided. For example, this expression returns the value `5-02`:

  ```
  INTERVAL '2-7' YEAR TO MONTH * 2
  ```

Examples of Interval Value Expressions

The PROJECT table consists of five columns using the data types NUMERIC, VARCHAR, DATE, TIMESTAMP, and INTERVAL DAY. Suppose that you have inserted values into the PROJECT table. For example:

```
INSERT INTO persnl.project
VALUES (1000,'SALT LAKE CITY',DATE '2007-04-10',
        TIMESTAMP '2007-04-21:08:15:00.00',INTERVAL '15' DAY);
```

The next example uses these values in the PROJECT table:

PROJCODE	START_DATE	SHIP_TIMESTAMP	EST_COMPLETE
1000	2007-04-10	2007-04-21:08:15:00.0000	15
2000	2007-06-10	2007-07-21:08:30:00.0000	30
2500	2007-10-10	2007-12-21:09:00:00.0000	60
3000	2007-08-21	2007-10-21:08:10:00.0000	60
4000	2007-09-21	2007-10-21:10:15:00.0000	30
5000	2007-09-28	2007-10-28:09:25:01.1111	30

- Suppose that the CURRENT_TIMESTAMP is 2000-01-06 11:14:41.748703. Find the number of days, hours, minutes, seconds, and fractional seconds in the difference of the current timestamp and the SHIP_TIMESTAMP in the PROJECT table:

  ```
  SELECT projcode,
    (CURRENT_TIMESTAMP - ship_timestamp) DAY(4) TO SECOND(6)
  FROM samdbcat.persnl.project;

  Project/Code  (EXPR)
  ------------  ---------------------
          1000   1355 02:58:57.087086
          2000   1264 02:43:57.087086
          2500   1111 02:13:57.087086
          3000   1172 03:03:57.087086
          4000   1172 00:58:57.087086
          5000   1165 01:48:55.975986

  --- 6 row(s) selected.
  ```

Numeric Value Expressions

- "Considerations for Numeric Value Expressions"
- "Examples of Numeric Value Expressions"

The operands of a numeric value expression can be combined in specific ways with arithmetic operators. In this syntax diagram, the data type of a term, factor, or numeric primary is numeric.

```
numeric-expression is:
     numeric-term
   | numeric-expression + numeric-term
   | numeric-expression - numeric-term

numeric-term is:
     numeric-factor
   | numeric-term * numeric-factor
   | numeric-term / numeric-factor

numeric-factor is:
     [+|-] numeric-primary
   | [+|-] numeric-primary ** numeric-factor

numeric-primary is:
     unsigned-numeric-literal
   | column-reference
   | numeric-type-host-variable
   | dynamic parameter
   | numeric-value-function
   | aggregate-function
   | sequence-function
   | scalar-subquery
   | CASE-expression
   | CAST-expression
   | (numeric-expression)
```

As shown in the preceding syntax diagram, numeric value expressions are built from operands that can be:

- Numeric literals
- Column references with numeric values
- Dynamic parameters
- Numeric value functions
- Aggregate functions, sequence functions, scalar subqueries, CASE expressions, or CAST expressions that return numeric values

Considerations for Numeric Value Expressions

Order of Evaluation

1. Expressions within parentheses
2. Unary operators
3. Exponentiation
4. Multiplication and division
5. Addition and subtraction

Operators at the same level are evaluated from left to right for all operators except exponentiation. Exponentiation operators at the same level are evaluated from right to left. For example, X + Y + Z is evaluated as (X + Y) + Z, whereas X ** Y ** Z is evaluated as X ** (Y ** Z).

Additional Rules for Arithmetic Operations

Numeric expressions are evaluated according to these additional rules:

- An expression with a numeric operator evaluates to null if any of the operands is null.
- Dividing by 0 causes an error.
- Exponentiation is allowed only with numeric data types. If the first operand is 0 (zero), the second operand must be greater than 0, and the result is 0. If the second operand is 0, the

first operand cannot be 0, and the result is 1. If the first operand is negative, the second operand must be a value with an exact numeric data type and a scale of zero.

- Exponentiation is subject to rounding error. In general, results of exponentiation should be considered approximate.

Precision, Magnitude, and Scale of Arithmetic Results

The precision, magnitude, and scale are computed during the evaluation of an arithmetic expression . Precision is the maximum number of digits in the expression. Magnitude is the number of digits to the left of the decimal point. Scale is the number of digits to the right of the decimal point.

For example, a column declared as NUMERIC (18, 5) has a precision of 18, a magnitude of 13, and a scale of 5. As another example, the literal 12345.6789 has a precision of 9, a magnitude of 5, and a scale of 4.

The maximum precision for exact numeric data types is 128 digits. The maximum precision for the REAL data type is approximately 7 decimal digits, and the maximum precision for the DOUBLE PRECISION data type is approximately 16 digits.

When Trafodion SQL encounters an arithmetic operator in an expression, it applies these rules (with the restriction that if the precision becomes greater than 18, the resulting precision is set to 18 and the resulting scale is the maximum of 0 and (18- (*resulted precision* - *resulted scale*)).

- If the operator is + or -, the resulting scale is the maximum of the scales of the operands. The resulting precision is the maximum of the magnitudes of the operands, plus the scale of the result, plus 1.
- If the operator is *, the resulting scale is the sum of the scales of the operands. The resulting precision is the sum of the magnitudes of the operands and the scale of the result.
- If the operator is /, the resulting scale is the sum of the scale of the numerator and the magnitude of the denominator. The resulting magnitude is the sum of the magnitude of the numerator and the scale of the denominator.

For example, if the numerator is NUMERIC (7, 3) and the denominator is NUMERIC (7, 5), the resulting scale is 3 plus 2 (or 5), and the resulting magnitude is 4 plus 5 (or 9). The expression result is NUMERIC (14, 5).

Conversion of Numeric Types for Arithmetic Operations

Trafodion SQL automatically converts between floating-point numeric types (REAL and DOUBLE PRECISION) and other numeric types. All numeric values in the expression are first converted to binary, with the maximum precision needed anywhere in the evaluation.

Examples of Numeric Value Expressions

These are examples of numeric value expressions:

`-57`	Numeric literal.
`salary * 1.10`	The product of the values in the SALARY column and a numeric literal.
`unit_price * qty_ordered`	The product of the values in the UNIT_PRICE and QTY_ORDERED columns.
`12 * (7 - 4)`	An expression whose operands are numeric literals.
`COUNT (DISTINCT city)`	Function applied to the values in a column.

Identifiers

SQL identifiers are names used to identify tables, views, columns, and other SQL entities. The two types of identifiers are regular and delimited. A delimited identifier is enclosed in double quotes ("). Case-insensitive delimited identifiers are used only for usernames and role names. Either regular, delimited, or case-sensitive delimited identifiers can contain up to 128 characters.

Regular Identifiers

Regular identifiers begin with a letter (A through Z and a through z), but can also contain digits (0 through 9) or underscore characters (_). Regular identifiers are not case-sensitive. You cannot use a reserved word as a regular identifier.

Delimited Identifiers

Delimited identifiers are character strings that appear within double quote characters (") and consist of alphanumeric characters, including the underscore character (_) or a dash (-). Unlike regular identifiers, delimited identifiers are case-sensitive. Trafodion SQL does not support spaces or special characters in delimited identifiers given the constraints of the underlying HBase filesystem. You can use reserved words as delimited identifiers.

Case-Insensitive Delimited Identifiers

Case-insensitive delimited identifiers, which are used for usernames and roles, are character strings that appear within double quote characters (") and consist of alphanumeric characters (A through Z and a through z), digits (0 through 9), underscores (_), dashes (-), periods (.), at symbols (@), and forward slashes (/), except for the leading at sign (@) or leading forward slash (/) character.

Unlike other delimited identifiers, case-insensitive-delimited identifiers are case-insensitive. Identifiers are up-shifted before being inserted into the SQL metadata. Thus, whether you specify a user's name as `"Penelope.Quan@hp.com"`, `"PENELOPE.QUAN@hp.com"`, or `"penelope.quan@hp.com"`, the value stored in the metadata will be the same: `PENELOPE.QUAN@HP.COM`.

You can use reserved words as case-insensitive delimited identifiers.

Examples of Identifiers

- These are regular identifiers:

  ```
  mytable
  SALES2006
  Employee_Benefits_Selections
  CUSTOMER_BILLING_INFORMATION
  ```

  Because regular identifiers are case insensitive, SQL treats all these identifiers as alternative representations of `mytable`:

  ```
  mytable         MYTABLE         MyTable         mYtAbLe
  ```

- These are delimited identifiers:

  ```
  "mytable"
  "table"
  "CUSTOMER-BILLING-INFORMATION"
  ```

  Because delimited identifiers are case-sensitive, SQL treats the identifier `"mytable"` as different from the identifiers `"MYTABLE"` or `"MyTable"`.

  You can use reserved words as delimited identifiers. For example, `table` is not allowed as a regular identifier, but `"table"` is allowed as a delimited identifier.

Indexes

An index is an ordered set of pointers to rows of a table. Each index is based on the values in one or more columns. Indexes are transparent to DML syntax.

A one-to-one correspondence always exists between index rows and base table rows.

SQL Indexes

Each row in a Trafodion SQL index contains:

- The columns specified in the CREATE INDEX statement
- The clustering key of the underlying table (the user-defined clustering key)

An index name is an SQL identifier. Indexes have their own namespace within a schema, so an index name might be the same as a table or constraint name. However, no two indexes in a schema can have the same name.

See "CREATE INDEX Statement" (page 53).

Keys

Trafodion SQL supports these types of keys:

- "Clustering Keys"
- "SYSKEY" (page 223)
- "Index Keys"
- "Primary Keys"

Clustering Keys

Every table has a clustering key, which is the set of columns that determine the order of the rows on disk. Trafodion SQL organizes records of a table or index by using a b-tree based on this clustering key. Therefore, the values of the clustering key act as logical row-ids.

SYSKEY

When the STORE BY clause is specified with the *key-column-list* clause, an additional column is appended to the *key-column-list* called the SYSKEY.

A SYSKEY (or system-defined clustering key) is a clustering key column which is defined by Trafodion SQL rather than by the user. Its type is LARGEINT SIGNED. When you insert a record in a table, Trafodion SQL automatically generates a value for the SYSKEY column. You cannot supply the value.

You cannot specify a SYSKEY at insert time and you cannot update it after it has been generated. To see the value of the generated SYSKEY, include the SYSKEY column in the select list:

```
SELECT *, SYSKEY FROM t4;
```

Index Keys

A one-to-one correspondence always exists between index rows and base table rows.

Each row in a Trafodion SQL index contains:

- The columns specified in the CREATE INDEX statement
- The clustering (primary) key of the underlying table (the user-defined clustering key)

For a nonunique index, the clustering key of the index is composed of both items. The clustering key cannot exceed 2048 bytes. Because the clustering key includes all the columns in the table, each row is also limited to 2048 bytes.

For varying-length character columns, the length referred to in these byte limits is the defined column length, not the stored length. (The stored length is the expanded length, which includes two extra bytes for storing the data length of the item.)

See "CREATE INDEX Statement" (page 53).

Primary Keys

A primary key is the column or set of columns that define the uniqueness constraint for a table. The columns cannot contain nulls, and only one primary key constraint can exist on a table.

Literals

A literal is a constant you can use in an expression, in a statement, or as a parameter value. An SQL literal can be one of these data types:

"Character String Literals" (page 224)	A series of characters enclosed in single quotes. Example: `'Planning'`
"Datetime Literals" (page 226)	Begins with keyword DATE, TIME, or TIMESTAMP and followed by a character string. Example: `DATE '1990-01-22'`
"Interval Literals" (page 227)	Begins with keyword INTERVAL and followed by a character string and an interval qualifier. Example: `INTERVAL '2-7' YEAR TO MONTH`
"Numeric Literals" (page 229)	A simple numeric literal (one without an exponent) or a numeric literal in scientific notation. Example: `99E-2`

Character String Literals

- "Considerations for Character String Literals"
- "Examples of Character String Literals"

A character string literal is a series of characters enclosed in single quotes.

You can specify either a string of characters or a set of hexadecimal code values representing the characters in the string.

```
[_character-set | N]'string'
| [_character-set | N] X'hex-code-value... '
| [_character-set | N] X'[space...]hex-code-value[[space...]hex-code-value...][space...]'
```

_character-set

specifies the character set ISO88591 or UTF8. The *_character-set* specification of the string literal should correspond with the character set of the column definition, which is either ISO88591 or UTF8. If you omit the _character-set specification, Trafodion SQL initially assumes the ISO88591 character set if the string literal consists entirely of 7-bit ASCII characters and UTF8 otherwise. (However, the initial assumption will later be changed if the string literal is used in a context that requires a character set different from the initial assumption.)

N

associates the string literal with the character set of the NATIONAL CHARACTER (NCHAR) data type. The character set for NCHAR is determined during the installation of Trafodion SQL. This value can be either UTF8 (the default) or ISO88591.

'*string*'

is a series of any input characters enclosed in single quotes. A single quote within a string is represented by two single quotes (''). A string can have a length of zero if you specify two single quotes ('') without a space in between.

X

indicates the hexadecimal string.

'*hex-code-value*'

represents the code value of a character in hexadecimal form enclosed in single quotes. It must contain an even number of hexadecimal digits. For ISO88591, each value must be two digits long. For UTF8, each value can be 2, 4, 6, or 8 hexadecimal digits long. If *hex-code-value* is improperly formatted (for example, it contains an invalid hexadecimal digit or an odd number of hexadecimal digits), an error is returned.

space

is space sequences that can be added before or after *hex-code-value* for readability. The encoding for *space* must be the TERMINAL_CHARSET for an interactive interface and the SQL module character set for the programmatic interface.

Considerations for Character String Literals

Using String Literals

A string literal can be as long as a character column. See "Character String Data Types" (page 204).

You can also use string literals in string value expressions—for example, in expressions that use the concatenation operator (||) or in expressions that use functions returning string values.

When specifying string literals:

- Do not put a space between the character set qualifier and the character string literal. If you use this character string literal in a statement, Trafodion SQL returns an error.
- To specify a single quotation mark within a string literal, use two consecutive single quotation marks.
- To specify a string literal whose length is more than one line, separate the literal into several smaller string literals, and use the concatenation operator (||) to concatenate them.
- Case is significant in string literals. Lowercase letters are not equivalent to the corresponding uppercase letters.
- Leading and trailing spaces within a string literal are significant.
- Alternately, a string whose length is more than one line can be written as a literal followed by a space, CR, or tab character, followed by another string literal.

Examples of Character String Literals

- These data type column specifications are shown with examples of literals that can be stored in the columns.

Character String Data Type	Character String Literal Example
CHAR (12) UPSHIFT	'PLANNING'
VARCHAR (18)	'NEW YORK'

- These are string literals:

```
'This is a string literal.'
'abc^&*'
'1234.56'
'This literal contains '' a single quotation mark.'
```

- This is a string literal concatenated over three lines:

```
'This literal is' ||
' in three parts,' ||
'specified over three lines.'
```

- This is a hexadecimal string literal representing the VARCHAR pattern of the ISO88591 string 'Strauß':

```
_ISO88591 X'53 74 72 61 75 DF'
```

Datetime Literals

- "Examples of Datetime Literals"

A datetime literal is a DATE, TIME, or TIMESTAMP constant you can use in an expression, in a statement, or as a parameter value. Datetime literals have the same range of valid values as the corresponding datetime data types. You cannot use leading or trailing spaces within a datetime string (within the single quotes).

A datetime literal begins with the `DATE`, `TIME`, or `TIMESTAMP` keyword and can appear in default, USA, or European format.

```
DATE 'date' | TIME 'time' | TIMESTAMP 'timestamp'

date is:
     yyyy-mm-dd                          Default
   | mm/dd/yyyy                          USA
   | dd.mm.yyyy                          European

time is:
     hh:mm:ss.mssssss                    Default
   | hh:mm:ss.mssssss [am | pm]          USA
   | hh.mm.ss.mssssss                    European

timestamp is:
     yyyy-mm-dd hh:mm:ss.mssssss         Default
   | mm/dd/yyyy hh:mm:ss.mssssss [am | pm]  USA
   | dd.mm.yyyy hh.mm.ss.mssssss         European
```

date,time,timestamp

specify the datetime literal strings whose component fields are:

`yyyy`	Year, from 0001 to 9999
`mm`	Month, from 01 to 12
`dd`	Day, from 01 to 31
`hh`	Hour, from 00 to 23
`mm`	Minute, from 00 to 59
`ss`	Second, from 00 to 59
`mssssss`	Microsecond, from 000000 to 999999
`am`	AM or am, indicating time from midnight to before noon
`pm`	PM or pm, indicating time from noon to before midnight

Examples of Datetime Literals

- These are DATE literals in default, USA, and European formats, respectively:

```
DATE '2008-01-22'
DATE '01/22/2008'
DATE '22.01.2008'
```

- These are TIME literals in default, USA, and European formats, respectively:

```
TIME '13:40:05'
TIME '01:40:05 PM'
TIME '13.40.05'
```

- These are TIMESTAMP literals in default, USA, and European formats, respectively:

```
TIMESTAMP '2008-01-22 13:40:05'
TIMESTAMP '01/22/2008 01:40:05 PM'
TIMESTAMP '22.01.2008 13.40.05'
```

Interval Literals

- "Considerations for Interval Literals"
- "Examples of Interval Literals"

An interval literal is a constant of data type INTERVAL that represents a positive or negative duration of time as a year-month or day-time interval; it begins with the keyword INTERVAL optionally preceded or followed by a minus sign (for negative duration). You cannot include leading or trailing spaces within an interval string (within single quotes).

```
[-]INTERVAL [-]{'year-month' | 'day:time'} interval-qualifier

year-month is:
   years [-months] | months

day:time is:
     days [[:]hours [:minutes [:seconds [.fraction]]]]
   | hours [:minutes [:seconds [.fraction]]]
   | minutes [:seconds [.fraction]]
   | seconds [.fraction]

interval-qualifier is:
   start-field TO end-field | single-field

start-field is:
   {YEAR | MONTH | DAY | HOUR | MINUTE} [(leading-precision)]

end-field is:
   YEAR | MONTH | DAY | HOUR | MINUTE | SECOND [(fractional-precision)]

single-field is:
   start-field | SECOND [(leading-precision,fractional-precision)]
```

start-field TO *end-field*

must be year-month or day-time.The *start-field* you specify must precede the *end-field* you specify in the list of field names.

{YEAR | MONTH | DAY | HOUR | MINUTE} [(*leading-precision*)]

specifies the *start-field*. A *start-field* can have a *leading-precision* up to 18 digits (the maximum depends on the number of fields in the interval). The *leading-precision* is the number of digits allowed in the *start-field*. The default for *leading-precision* is 2.

YEAR | MONTH | DAY | HOUR | MINUTE | SECOND [(*fractional-precision*)]

specifies the *end-field*. If the *end-field* is SECOND, it can have a *fractional-precision* up to 6 digits. The *fractional-precision* is the number of digits of precision after the decimal point. The default for *fractional-precision* is 6.

start-field | SECOND [(*leading-precision*, *fractional-precision*)]

specifies the *single-field*. If the *single-field* is SECOND, the *leading-precision* is the number of digits of precision before the decimal point, and the *fractional-precision* is the number of digits of precision after the decimal point.

The default for *leading-precision* is 2, and the default for *fractional-precision* is 6. The maximum for *leading-precision* is 18, and the maximum for *fractional-precision* is 6.

See "Interval Data Types" (page 207) and "Interval Value Expressions" (page 215).

'*year-month*' | '*day:time*'

specifies the date and time components of an interval literal. The day and hour fields can be separated by a space or a colon. The interval literal strings are:

years	Unsigned integer that specifies a number of years. *years* can be up to 18 digits, or 16 digits if *months* is the end-field. The maximum for the *leading-precision* is specified within the interval qualifier by either YEAR(18) or YEAR(16) TO MONTH.
months	Unsigned integer that specifies a number of months. Used as a starting field, *months* can have up to 18 digits. The maximum for the *leading-precision* is specified by MONTH(18). Used as an ending field, the value of *months* must be in the range 0 to 11.
days	Unsigned integer that specifies number of days. *days* can have up to 18 digits if no end-field exists; 16 digits if *hours* is the end-field; 14 digits if *minutes* is the end-field; and 13-*f* digits if *seconds* is the end-field, where f is the *fraction* less than or equal to 6. These maximums are specified by DAY(18), DAY(16) TO HOUR, DAY(14) TO MINUTE, and DAY(13-*f*) TO SECOND(*f*).
hours	Unsigned integer that specifies a number of hours. Used as a starting field, *hours* can have up to 18 digits if no end-field exists; 16 digits if *minutes* is the end-field; and 14-*f* digits if *seconds* is the end-field, where f is the *fraction* less than or equal to 6. These maximums are specified by HOUR(18), HOUR(16) TO MINUTE, and HOUR(14-f) TO SECOND(*f*). Used as an ending field, the value of *hours* must be in the range 0 to 23.
minutes	Unsigned integer that specifies a number of minutes. Used as a starting field, *minutes* can have up to 18 digits if no end-field exists; and 16-f digits if *seconds* is the end-field, where *f* is the *fraction* less than or equal to 6. These maximums are specified by MINUTE(18), and MINUTE(16-*f*) TO SECOND(*f*). Used as an ending field, the value of *minutes* must be in the range 0 to 59.
seconds	Unsigned integer that specifies a number of seconds. Used as a starting field, *seconds* can have up to 18 digits, minus the number of digits f in the *fraction* less than or equal to 6. This maximum is specified by SECOND(18-*f*, *f*). The value of *seconds* must be in the range 0 to 59.9(*n*), where *n* is the number of digits specified for seconds precision.
fraction	Unsigned integer that specifies a fraction of a second. When *seconds* is used as an ending field, *fraction* is limited to the number of digits specified by the *fractional-precision* field following the SECOND keyword.

Considerations for Interval Literals

Length of Year-Month and Day-Time Strings

An interval literal can contain a maximum of 18 digits, in the string following the INTERVAL keyword, plus a hyphen (-) that separates the year-month fields, and colons (:) that separate the day-time fields. You can also separate day and hour with a space.

Examples of Interval Literals

`INTERVAL '1' MONTH`	Interval of 1 month
`INTERVAL '7' DAY`	Interval of 7 days
`INTERVAL '2-7' YEAR TO MONTH`	Interval of 2 years, 7 months
`INTERVAL '5:2:15:36.33' DAY TO SECOND(2)`	Interval of 5 days, 2 hours, 15 minutes, and 36.33 seconds
`INTERVAL - '5' DAY`	Interval that subtracts 5 days
`INTERVAL '100' DAY(3)`	Interval of 100 days. This example requires an explicit leading precision of 3 because the default is 2.
`INTERVAL '364 23' DAY(3) TO HOUR`	Interval of 364 days, 23 hours. The separator for the day and hour fields can be a space or a colon.

Numeric Literals

A numeric literal represents a numeric value. Numeric literals can be represented as an exact numeric literal (without an exponent) or as an approximate numeric literal by using scientific notation (with an exponent).

```
exact-numeric-literal is:
      [+|-]unsigned-integer[.[unsigned-integer]]
   |  [+|-].unsigned-integer

approximate-numeric-literal is:
   mantissa{E|e}exponent

mantissa is:
   exact-numeric-literal

exponent is:
   [+|-]unsigned-integer

unsigned-integer is:
   digit...
```

exact-numeric-literal

is an exact numeric value that includes an optional plus sign (+) or minus sign (-), up to 128 digits (0 through 9), and an optional period (.) that indicates a decimal point. Leading zeros do not count toward the 128-digit limit; trailing zeros do.

A numeric literal without a sign is a positive number. An exact numeric literal that does not include a decimal point is an integer. Every exact numeric literal has the data type NUMERIC and the minimum precision required to represent its value.

approximate-numeric-literal

is an exact numeric literal followed by an exponent expressed as an uppercase E or lowercase e followed by an optionally signed integer.

Numeric values expressed in scientific notation are treated as data type REAL if they include no more than seven digits before the exponent, but treated as type DOUBLE PRECISION if they include eight or more digits. Because of this factor, trailing zeros after a decimal can sometimes increase the precision of a numeric literal used as a DOUBLE PRECISION value.

For example, if XYZ is a table that consists of one DOUBLE PRECISION column, the inserted value:

```
INSERT INTO XYZ VALUES (1.00000000E-10)
```

has more precision than:

```
INSERT INTO XYZ VALUES (1.0E-10)
```

Examples of Numeric Literals

These are all numeric literals, along with their display format:

Literal	Display Format
477	477
580.45	580.45
+005	5
-.3175	-.3175
1300000000	1300000000
99.	99
-0.123456789012345678	-.123456789012345678
99E-2	9.9000000E-001
12.3e+5	1.2299999E+006

Null

Null is a special symbol, independent of data type, that represents an unknown. The Trafodion SQL keyword NULL represents null. Null indicates that an item has no value. For sorting purposes, null is greater than all other values. You cannot store null in a column by using INSERT or UPDATE, unless the column allows null.

A column that allows null can be null at any row position. A nullable column has extra bytes associated with it in each row. A special value stored in these bytes indicates that the column has null for that row.

Using Null Versus Default Values

Various scenarios exist in which a row in a table might contain no value for a specific column. For example:

- A database of telemarketing contacts might have null AGE fields if contacts did not provide their age.
- An order record might have a DATE_SHIPPED column empty until the order is actually shipped.
- An employee record for an international employee might not have a social security number.

You allow null in a column when you want to convey that a value in the column is unknown (such as the age of a telemarketing contact) or not applicable (such as the social security number of an international employee).

In deciding whether to allow nulls or use defaults, also note:

- Nulls are not the same as blanks. Two blanks can be compared and found equal, while the result of a comparison of two nulls is indeterminate.
- Nulls are not the same as zeros. Zeros can participate in arithmetic operations, while nulls are excluded from any arithmetic operation.

Defining Columns That Allow or Prohibit Null

The CREATE TABLE and ALTER TABLE statements define the attributes for columns within tables. A column allows nulls unless the column definition includes the NOT NULL clause or the column is part of the primary key of the table.

Null is the default for a column (other than NOT NULL) unless the column definition includes a DEFAULT clause (other than DEFAULT NULL) or the NO DEFAULT clause. The default value for a column is the value Trafodion SQL inserts in a row when an INSERT statement omits a value for a particular column.

Null in DISTINCT, GROUP BY, and ORDER BY Clauses

In evaluating the DISTINCT, GROUP BY, and ORDER BY clauses, Trafodion SQL considers all nulls to be equal. Additional considerations for these clauses are:

DISTINCT	Nulls are considered duplicates; therefore, a result has at most one null.
GROUP BY	The result has at most one null group.
ORDER BY	Nulls are considered greater than nonnull values.

Null and Expression Evaluation Comparison

Expression Type	Condition	Result
Boolean operators (AND, OR, NOT)	Either operand is null.	For AND, the result is null. For OR, the result is true if the other operand is true, or null if the other operand is null or false. For NOT, the result is null.
Arithmetic operators	Either or both operands are null.	The result is null.
NULL predicate	The operand is null.	The result is true.
Aggregate (or set) functions (except COUNT)	Some rows have null columns.The function is evaluated after eliminating nulls.	The result is null if set is empty.
COUNT(*)	The function does not eliminate nulls.	The result is the number of rows in the table whether or not the rows are null.
COUNT COUNT DISTINCT	The function is evaluated after eliminating nulls.	The result is zero if set is empty.
Comparison: =, <>, <, >, <=, >=, LIKE	Either operand is null.	The result is null.
IN predicate	Some expressions in the IN value list are null.	The result is null if all of the expressions are null.
Subquery	No rows are returned.	The result is null.

Predicates

A predicate determines an answer to a question about a value or group of values. A predicate returns true, false, or, if the question cannot be answered, unknown. Use predicates within search conditions to choose rows from tables or views.

"BETWEEN Predicate" (page 233)	Determines whether a sequence of values is within a range of sequences of values.
"Comparison Predicates" (page 234) (=, <>, <, >, <=, >=)	Compares the values of sequences of expressions, or compares the values of sequences of row values that are the result of row subqueries.
"EXISTS Predicate" (page 238)	Determines whether any rows are selected by a subquery. If the subquery finds at least one row that satisfies its search condition, the predicate evaluates to true. Otherwise, if the result table of the subquery is empty, the predicate is false.
"IN Predicate" (page 239)	Determines if a sequence of values is equal to any of the sequences of values in a list of sequences.
"LIKE Predicate" (page 241)	Searches for character strings that match a pattern.
"NULL Predicate" (page 243)	Determines whether all the values in a sequence of values are null.
"Quantified Comparison Predicates" (page 244) (ALL, ANY, SOME)	Compares the values of sequences of expressions to the values in each row selected by a table subquery. The comparison is quantified by ALL, SOME, or ANY.

See the individual entry for a predicate or predicate group.

BETWEEN Predicate

- "Considerations for BETWEEN"
- "Examples of BETWEEN"

The BETWEEN predicate determines whether a sequence of values is within a range of sequences of values.

```
row-value-constructor [NOT] BETWEEN
    row-value-constructor AND row-value-constructor
row-value-constructor is:
     (expression [,expression ]...)
   | row-subquery
```

row-value-constructor

specifies an operand of the BETWEEN predicate. The three operands can be either of:

(*expression* [,*expression*]...)

is a sequence of SQL value expressions, separated by commas and enclosed in parentheses. *expression* cannot include an aggregate function unless *expression* is in a HAVING clause. *expression* can be a scalar subquery (a subquery that returns a single row consisting of a single column). See "Expressions" (page 211).

row-subquery

is a subquery that returns a single row (consisting of a sequence of values). See "Subquery" (page 252).

The three *row-value-constructors* specified in a BETWEEN predicate must contain the same number of elements. That is, the number of value expressions in each list, or the number of values returned by a row subquery, must be the same.

The data types of the respective values of the three *row-value-constructors* must be comparable. Respective values are values with the same ordinal position in the two lists. See "Comparable and Compatible Data Types" (page 201).

Considerations for BETWEEN

Logical Equivalents Using AND and OR

The predicate *expr1* BETWEEN *expr2* AND *expr3* is true if and only if this condition is true:

```
expr2 <= expr1 AND expr1 <= expr3
```

The predicate *expr1* NOT BETWEEN *expr2* AND *expr3* is true if and only if this condition is true:

```
expr2 > expr1 OR expr1 > expr3
```

Descending Columns in Keys

If a clause specifies a column in a key BETWEEN *expr2* and *expr3*, *expr3* must be greater than *expr2* even if the column is specified as DESCENDING within its table definition.

Examples of BETWEEN

- This predicate is true if the total price of the units in inventory is in the range from $1,000 to $10,000:

```
qty_on_hand * price
  BETWEEN 1000.00 AND 10000.00
```

- This predicate is true if the part cost is less than $5 or more than $800:

```
partcost NOT BETWEEN 5.00 AND 800.00
```

- This BETWEEN predicate selects the part number 6400:

```
SELECT * FROM partsupp
WHERE partnum BETWEEN 6400 AND 6700
  AND partcost > 300.00;

Part/Num  Supp/Num  Part/Cost     Qty/Rec
--------  --------  ------------  ----------
    6400         1        390.00          50
    6401         2        500.00          20
    6401         3        480.00          38

--- 3 row(s) selected.
```

- Find names between Jody Selby and Gene Wright:

```
(last_name, first_name) BETWEEN
  ('SELBY', 'JODY') AND ('WRIGHT', 'GENE')
```

The name Barbara Swift would meet the criteria; the name Mike Wright would not.

```
SELECT empnum, first_name, last_name
FROM persnl.employee
WHERE (last_name, first_name) BETWEEN
  ('SELBY', 'JODY') AND ('WRIGHT', 'GENE');

EMPNUM  FIRST_NAME       LAST_NAME
------  ---------------  --------------------
    43  PAUL             WINTER
    72  GLENN            THOMAS
    74  JOHN             WALKER
   ...
--- 15 row(s) selected.
```

Comparison Predicates

- "Considerations for Comparison Predicates"
- "Examples of Comparison Predicates"

A comparison predicate compares the values of sequences of expressions, or the values of sequences of row values that are the result of row subqueries.

```
row-value-constructor comparison-op row-value-constructor

comparison-op is:
     =    Equal
   | <>   Not equal
   | <    Less than
   | >    Greater than
   | <=   Less than or equal to
   | >=   Greater than or equal to

row-value-constructor is:
     (expression [,expression]...)
   | row-subquery
```

row-value-constructor

specifies an operand of a comparison predicate. The two operands can be either of these:

(*expression* [,*expression*]...)

is a sequence of SQL value expressions, separated by commas and enclosed in parentheses. *expression* cannot include an aggregate function unless expression is in a HAVING clause. *expression* can be a scalar subquery (a subquery that returns a single row consisting of a single column). See "Expressions" (page 211).

row-subquery

is a subquery that returns a single row (consisting of a sequence of values). See "Subquery" (page 252).

The two *row-value-constructors* must contain the same number of elements. That is, the number of value expressions in each list, or the number of values returned by a row subquery, must be the same.

The data types of the respective values of the two *row-value-constructors* must be comparable. (Respective values are values with the same ordinal position in the two lists.) See "Comparable and Compatible Data Types" (page 201).

Considerations for Comparison Predicates

When a Comparison Predicate Is True

Trafodion SQL determines whether a relationship is true or false by comparing values in corresponding positions in sequence, until it finds the first nonequal pair.

You cannot use a comparison predicate in a WHERE or HAVING clause to compare row value constructors when the value expressions in one row value constructor are equal to null. Use the IS NULL predicate instead.

Suppose that two rows with multiple components exist, X and Y:

```
X=(X1,X2,...,Xn), Y=(Y1,Y2,...,Yn).
```

Predicate `X=Y` is true if for all `i=1,...,n: Xi=Yi`. For this predicate, Trafodion SQL must look through all values. Predicate `X = Y` is false if for some `i Xi<>Yi`. When SQL finds nonequal components, it stops and does not look at remaining components.

Predicate `X<>Y` is true if `X=Y` is false. If `X1<>Y1`, Trafodion SQL does not look at all components. It stops and returns a value of false for the `X=Y` predicate and a value of true for the `X<>Y` predicate. Predicate `X<>Y` is false if `X=Y` is true, or for all `i=1,...,n: Xi=Yi`. In this situation, Trafodion SQL must look through all components.

Predicate `X>Y` is true if for some index `m Xm>Ym` and for all `i=1,..,m-1: Xi=Yi`. Trafodion SQL does not look through all components. It stops when it finds the first nonequal components, `Xm<>Ym`. If `Xm>Ym`, the predicate is true. Otherwise the predicate is false. The predicate is also false if all components are equal, or `X=Y`.

Predicate `X>=Y` is true if `X>Y` is true or `X=Y` is true. In this scenario, Trafodion SQL might look through all components and return true if they are all equal. It stops at the first nonequal components, `Xm<>Ym`. If `Xm>Ym`, the predicate is true. Otherwise, it is false.

Predicate `X<Y` is true if for some index `m Xm<Ym`, and for all `i=1,..,m-1: Xi=Yi`. Trafodion SQL does not look through all components. It stops when it finds the first nonequal components `Xm<>Ym`. If `Xm<Ym`, the predicate is true. Otherwise, the predicate is false. The predicate is also false if all components are equal, or `X=Y`.

Predicate `X<=Y` is true if `X<Y` is true or `X=Y` is true. In this scenario, Trafodion SQL might need to look through all components and return true if they are all equal. It stops at the first nonequal components, `Xm<>Ym`. If `Xm<Ym`, the predicate is true. Otherwise, it is false.

Comparing Character Data

For comparisons between character strings of different lengths, the shorter string is padded on the right with spaces (HEX 20) until it is the length of the longer string. Both fixed-length and variable-length strings are padded in this way.

For example, Trafodion SQL considers the string 'JOE' equal to a value JOE stored in a column of data type CHAR or VARCHAR of width three or more. Similarly, Trafodion SQL considers a value JOE stored in any column of the CHAR data type equal to the value JOE stored in any column of the VARCHAR data type.

Two strings are equal if all characters in the same ordinal position are equal. Lowercase and uppercase letters are not considered equivalent.

Comparing Numeric Data

Before evaluation, all numeric values in an expression are first converted to the maximum precision needed anywhere in the expression.

Comparing Interval Data

For comparisons of INTERVAL values, Trafodion SQL first converts the intervals to a common unit. If no common unit exists, Trafodion SQL reports an error. Two INTERVAL values must be both year-month intervals or both day-time intervals.

Comparing Multiple Values

Use multivalue predicates whenever possible; they are generally more efficient than equivalent conditions without multivalue predicates.

Examples of Comparison Predicates

- This predicate is true if the customer number is equal to 3210:

  ```
  custnum = 3210
  ```

- This predicate is true if the salary is greater than the average salary of all employees:

  ```
  salary >
    (SELECT AVG (salary) FROM persnl.employee);
  ```

- This predicate is true if the customer name is BACIGALUPI:

  ```
  custname = 'BACIGALUPI'
  ```

- This predicate evaluates to unknown for any rows in either CUSTOMER or ORDERS that contain null in the CUSTNUM column:

  ```
  customer.custnum > orders.custnum
  ```

- This predicate returns information about anyone whose name follows MOSS, DUNCAN in a list arranged alphabetically by last name and, for the same last name, alphabetically by first name:

  ```
  (last_name, first_name) > ('MOSS', 'DUNCAN')
  ```

  REEVES, ANNE meets this criteria, but MOSS, ANNE does not.

This multivalue predicate is equivalent to this condition with three comparison predicates:

```
(last_name > 'MOSS') OR
(last_name = 'MOSS' AND first_name > 'DUNCAN')
```

- Compare two datetime values START_DATE and the result of the CURRENT_DATE function:

  ```
  START_DATE < CURRENT_DATE
  ```

- Compare two datetime values START_DATE and SHIP_TIMESTAMP:

  ```
  CAST (start_date AS TIMESTAMP) < ship_timestamp
  ```

- Compare two INTERVAL values:

  ```
  JOB1_TIME < JOB2_TIME
  ```

  Suppose that JOB1_TIME, defined as INTERVAL DAY TO MINUTE, is 2 days 3 hours, and JOB2_TIME, defined as INTERVAL DAY TO HOUR, is 3 days.

  To evaluate the predicate, Trafodion SQL converts the two INTERVAL values to MINUTE. The comparison predicate is true.

- The next examples contain a subquery in a comparison predicate. Each subquery operates on a separate logical copy of the EMPLOYEE table.

  The processing sequence is outer to inner. A row selected by an outer query allows an inner query to be evaluated, and a single value is returned. The next inner query is evaluated when it receives a value from its outer query.

  Find all employees whose salary is greater than the maximum salary of employees in department 1500:

  ```
  SELECT first_name, last_name, deptnum, salary
    FROM persnl.employee
    WHERE salary > (SELECT MAX (salary)
                    FROM persnl.employee
                    WHERE deptnum = 1500);

  FIRST_NAME       LAST_NAME            DEPTNUM  SALARY
  ---------------  -------------------  -------  -----------
  ROGER            GREEN                   9000    175500.00
  KATHRYN          HALL                    4000     96000.00
  RACHEL           MCKAY                   4000    118000.00
  THOMAS           RUDLOFF                 2000    138000.40
  JANE             RAYMOND                 3000    136000.00
  JERRY            HOWARD                  1000    137000.10

  --- 6 row(s) selected.
  ```

  Find all employees from other departments whose salary is less than the minimum salary of employees (not in department 1500) that have a salary greater than the average salary for department 1500:

  ```
  SELECT first_name, last_name, deptnum, salary
  FROM persnl.employee
  WHERE deptnum <> 1500 AND
    salary < (SELECT MIN (salary)
              FROM persnl.employee
              WHERE deptnum <> 1500 AND
                salary > (SELECT AVG (salary)
                          FROM persnl.employee
                          WHERE deptnum = 1500));

  FIRST_NAME       LAST_NAME            DEPTNUM  SALARY
  ---------------  -------------------  -------  -----------
  JESSICA          CRINER                  3500     39500.00
  ALAN             TERRY                   3000     39500.00
  DINAH            CLARK                   9000     37000.00
  ```

```
BILL            WINN                    2000     32000.00
MIRIAM          KING                    2500     18000.00
...

--- 35 row(s) selected.
```

The first subquery of this query determines the minimum salary of employees from other departments whose salary is greater than the average salary for department 1500. The main query then finds the names of employees who are not in department 1500 and whose salary is less than the minimum salary determined by the first subquery.

EXISTS Predicate

The EXISTS predicate determines whether any rows are selected by a subquery. If the subquery finds at least one row that satisfies its search condition, the predicate evaluates to true. Otherwise, if the result table of the subquery is empty, the predicate is false.

```
[NOT] EXISTS subquery
```

subquery

specifies the operand of the predicate. A *subquery* is a query expression enclosed in parentheses. An EXISTS *subquery* is typically correlated with an outer query. See "Subquery" (page 252).

Examples of EXISTS

- Find locations of employees with job code 300:

```
SELECT deptnum, location FROM persnl.dept D
WHERE EXISTS
 (SELECT jobcode FROM persnl.employee E
  WHERE D.deptnum = E.deptnum AND jobcode = 300);

DEPTNUM  LOCATION
-------  -------------
   3000  NEW YORK
   3100  TORONTO
   3200  FRANKFURT
   3300  LONDON
   3500  HONG KONG

--- 5 row(s) selected.
```

In the preceding example, the EXISTS predicate contains a subquery that determines which locations have employees with job code 300. The subquery depends on the value of D.DEPTNUM from the outer query and must be evaluated for each row of the result table where D.DEPTNUM equals E.DEPTNUM. The column D.DEPTNUM is an example of an outer reference.

- Search for departments that have no employees with job code 420:

```
SELECT deptname FROM persnl.dept D
WHERE NOT EXISTS
  (SELECT jobcode FROM persnl.employee E
   WHERE D.deptnum = E.deptnum AND jobcode = 420);

DEPTNAME
------------
FINANCE
PERSONNEL
INVENTORY
...
--- 11 row(s) selected.
```

- Search for parts with less than 20 units in the inventory:

```
SELECT partnum, suppnum
FROM invent.partsupp PS
WHERE EXISTS
  (SELECT partnum FROM invent.partloc PL
   WHERE PS.partnum = PL.partnum AND qty_on_hand < 20);

PARTNUM  SUPPNUM
-------  -------
    212        1
    212        3
   2001        1
   2003        2
    ...

--- 18 row(s) selected.
```

IN Predicate

- "Considerations for IN"
- "Examples of IN"

The IN predicate determines if a sequence of values is equal to any of the sequences of values in a list of sequences. The NOT operator reverses its truth value. For example, if IN is true, NOT IN is false.

```
row-value-constructor
   [NOT] IN {table-subquery | in-value-list}

row-value-constructor is:
     (expression [,expression]...)
   | row-subquery

in-value-list is:
   (expression [,expression]...)
```

row-value-constructor

specifies the first operand of the IN predicate. The first operand can be either of:

(*expression* [,*expression*]...)

is a sequence of SQL value expressions, separated by commas and enclosed in parentheses. *expression* cannot include an aggregate function unless expression is in a HAVING clause. *expression* can be a scalar subquery (a subquery that returns a single row consisting of a single column). See "Expressions" (page 211).

row-subquery

is a subquery that returns a single row (consisting of a sequence of values). See "Subquery" (page 252).

table-subquery

is a subquery that returns a table (consisting of rows of columns). The table specifies rows of values to be compared with the row of values specified by the *row-value-constructor*. The number of values of the *row-value-constructor* must be equal to the number of columns in the result table of the *table-subquery*, and the data types of the values must be comparable.

in-value-list

is a sequence of SQL value expressions, separated by commas and enclosed in parentheses. *expression* cannot include an aggregate function defined on a column. *expression* can be a scalar subquery (a subquery that returns a single row consisting of a single column). In this case, the result of the *row-value-constructor* is a single value. The data types of the values must be comparable. The number of expressions in the *in-value-list* can have at least 5000 expressions.

Considerations for IN

Logical Equivalent Using ANY (or SOME)

The predicate *expr* `IN (`*expr1*`, `*expr2*`, ... )` is true if and only if the following predicate is true:

expr `= ANY (`*expr1*`, `*expr2*`, ... )`

IN Predicate Results

The IN predicate is true if and only if either of these is true:

- The result of the *row-value-constructor* (a row or sequence of values) is equal to any row of column values specified by *table-subquery*.

  A table subquery is a query expression and can be specified as a form of a simple table; for example, as the VALUES keyword followed by a list of row values. See "SELECT Statement" (page 138).

- The result of the *row-value-constructor* (a single value) is equal to any of the values specified by the list of expressions *in-value-list*.

  In this case, it is helpful to think of the list of expressions as a one-column table—a special case of a table subquery. The degree of the row value constructor and the degree of the list of expressions are both one.

Comparing Character Data

Two strings are equal if all characters in the same ordinal position are equal. Lowercase and uppercase letters are not considered equivalent. For comparisons between character strings of different lengths, the shorter string is padded on the right with spaces (HEX 20) until it is the length of the longer string. Both fixed-length and varying-length strings are padded in this way.

For example, Trafodion SQL considers the string 'JOE' equal to a value JOE stored in a column of data type CHAR or VARCHAR of width three or more. Similarly, Trafodion SQL considers a value JOE stored in any column of the CHAR data type equal to the value JOE stored in any column of the VARCHAR data type.

Comparing Numeric Data

Before evaluation, all numeric values in an expression are first converted to the maximum precision needed anywhere in the expression.

Comparing Interval Data

For comparisons of INTERVAL values, Trafodion SQL first converts the intervals to a common unit. If no common unit exists, Trafodion SQL reports an error. Two INTERVAL values must be both year-month intervals or both day-time intervals.

Examples of IN

- Find those employees whose EMPNUM is 39, 337, or 452:

```
SELECT last_name, first_name, empnum
FROM persnl.employee
WHERE empnum IN (39, 337, 452);

LAST_NAME             FIRST_NAME       EMPNUM
--------------------  ---------------  ------
CLARK                 DINAH               337
SAFFERT               KLAUS                39
--- 2 row(s) selected.
```

- Find those items in PARTS whose part number is not in the PARTLOC table:

```
SELECT partnum, partdesc
FROM sales.parts
WHERE partnum NOT IN
  (SELECT partnum
```

```
   FROM invent.partloc);

PARTNUM  PARTDESC
-------  ------------------
    186  186 MegaByte Disk

--- 1 row(s) selected.
```

- Find those items (and their suppliers) in PARTS that have a supplier in the PARTSUPP table:

```
SELECT P.partnum, P.partdesc, S.suppnum, S.suppname
FROM sales.parts P,
  invent.supplier S
WHERE P.partnum, S.suppnum IN
  (SELECT partnum, suppnum
   FROM invent.partsupp);
```

- Find those employees in EMPLOYEE whose last name and job code match the list of last names and job codes:

```
SELECT empnum, last_name, first_name
FROM persnl.employee
WHERE (last_name, jobcode) IN
  (VALUES ('CLARK', 500), ('GREEN', 200));
```

LIKE Predicate

The LIKE predicate searches for character strings that match a pattern.

Syntax

```
match-value [NOT] LIKE pattern [ESCAPE esc-char-expression]
```

match-value
is a character value expression that specifies a set of strings to search for that match the *pattern*.

pattern
is a character value expression that specifies the pattern string for the search.

esc-char-expression
is a character value expression that must evaluate to a single character. The escape character value is used to turn off the special meaning of percent (%) and underscore (_). See "Wild-Card Characters" (page 242) and "Escape Characters" (page 242).

See "Character Value Expressions" (page 211).

Considerations

Comparing the Value to the Pattern

The values that you compare must be character strings. Lowercase and uppercase letters are not equivalent. To make lowercase letters match uppercase letters, use the UPSHIFT function. A blank is compared in the same way as any other character.

When a LIKE Predicate Is True

When you refer to a column, the LIKE predicate is true if the *pattern* matches the column value. If the value of the column reference is null, the LIKE predicate evaluates to unknown for that row. If the values that you compare are both empty strings (that is, strings of zero length), the LIKE predicate is true.

Using NOT

If you specify NOT, the predicate is true if the *pattern* does not match any string in the *match-value* or is not the same length as any string in the *match-value*. For example, NAME NOT LIKE '_Z' is true if the string is not two characters long or the last character is not Z. In a search condition, the predicate NAME NOT LIKE '_Z' is equivalent to NOT (NAME LIKE '_Z').

Wild-Card Characters

You can look for similar values by specifying only part of the characters of *pattern* combined with these wild-card characters:

- "Percent Sign (%)" (page 242)
- "Underscore (_)" (page 242)

Percent Sign (%)

Use a percent sign to indicate zero or more characters of any type. For example, '%ART%' matches 'SMART', 'ARTIFICIAL', and 'PARTICULAR', but not 'smart'.

Underscore (_)

Use an underscore to indicate any single character. For example, 'BOO_' matches 'BOOK' and 'BOOT' but not 'BOO', 'BOOKLET', or 'book'.

Escape Characters

To search for a string containing a percent sign (%) or an underscore (_), define an escape character, using ESCAPE *esc-char-expression*, to turn off the special meaning of the percent sign and underscore.

To include a percent sign or an underscore in a comparison string, type the escape character immediately preceding the percent sign or underscore. For example, to locate the value 'A_B', type:

```
NAME LIKE 'A\_B' ESCAPE '\'
```

To include the escape character itself in the comparison string, type two escape characters. For example, to locate 'A_B\C%', type:

```
NAME LIKE 'A\_B\\C\%' ESCAPE '\'
```

The escape character must precede only the percent sign, underscore, or escape character itself. For example, the pattern RA\BS is an invalid LIKE pattern if the escape character is defined to be '\'. Error 8410 will be returned if this kind of pattern is used in an SQL query.

Comparing the Pattern to CHAR Columns

Columns of data type CHAR are fixed length. When a value is inserted into a CHAR column, Trafodion SQL pads the value in the column with blanks if necessary. The value 'JOE' inserted into a CHAR(4) column becomes 'JOE ' (three characters plus one blank). The LIKE predicate is true only if the column value and the comparison value are the same length. The column value 'JOE ' does not match 'JOE' but does match 'JOE%'.

Comparing the Pattern to VARCHAR Columns

Columns of variable-length character data types do not include trailing blanks unless blanks are specified when data is entered. For example, the value 'JOE' inserted in a VARCHAR(4) column is 'JOE' with no trailing blanks. The value matches both 'JOE' and 'JOE%'.

If you cannot locate a value in a variable-length character column, it might be because trailing blanks were specified when the value was inserted into the table. For example, a value of '5MB ' (with one trailing blank) will not be located by LIKE '%MB' but will be located by LIKE '%MB%'.

Examples

- Find all employee last names beginning with ZE:

  ```
  last_name LIKE 'ZE%'
  ```

- Find all part descriptions that are not 'FLOPPY_DISK':

  ```
  partdesc NOT LIKE 'FLOPPY\_DISK' ESCAPE '\'
  ```

  The escape character indicates that the underscore in 'FLOPPY_DISK' is part of the string to search for, not a wild-card character.

NULL Predicate

The NULL predicate determines whether all the expressions in a sequence are null. See "Null" (page 231).

```
row-value-constructor IS [NOT] NULL

row-value-constructor is:
      (expression [,expression]...)
    | row-subquery
```

row-value-constructor

specifies the operand of the NULL predicate. The operand can be either of these:

(*expression* [,*expression*]...)

is a sequence of SQL value expressions, separated by commas and enclosed in parentheses. *expression* cannot include an aggregate function unless expression is in a HAVING clause. *expression* can be a scalar subquery (a subquery that returns a single row consisting of a single column). See "Expressions" (page 211).

row-subquery

is a subquery that returns a single row (consisting of a sequence of values). See "Subquery" (page 252).

If all of the expressions in the *row-value-constructor* are null, the IS NULL predicate is true. Otherwise, it is false. If none of the expressions in the *row-value-constructor* are null, the IS NOT NULL predicate is true. Otherwise, it is false.

Considerations for NULL

Summary of NULL Results

Let rvc be the value of the *row-value-constructor*. This table summarizes the results of NULL predicates. The degree of a *rvc* is the number of values in the *rvc*.

Expressions	*rvc* IS NULL	*rvc* IS NOT NULL	NOT *rvc*IS NULL	NOT *rvc* IS NOT NULL
degree 1: null	TRUE	FALSE	FALSE	TRUE
degree 1: not null	FALSE	TRUE	TRUE	FALSE
degree>1: all null	TRUE	FALSE	FALSE	TRUE
degree>1: some null	FALSE	FALSE	TRUE	TRUE
degree>1: none null	FALSE	TRUE	TRUE	FALSE

The *rvc* IS NOT NULL predicate is not equivalent to NOT *rvc* IS NULL.

Examples of NULL

- Find all rows with null in the SALARY column:

```
salary IS NULL
```

- This predicate evaluates to true if the expression (PRICE + TAX) evaluates to null:

```
(price + tax) IS NULL
```

- Find all rows where both FIRST_NAME and SALARY are null:

```
(first_name, salary) IS NULL
```

Quantified Comparison Predicates

- "Considerations for ALL, ANY, SOME"
- "Examples of ALL, ANY, SOME"

A quantified comparison predicate compares the values of sequences of expressions to the values in each row selected by a table subquery. The comparison operation is quantified by the logical quantifiers ALL, ANY, or SOME.

```
row-value-constructor comparison-op quantifier table-subquery

row-value-constructor is:
     (expression [,expression]...)
   | row-subquery

comparison-op is:
     =    Equal
   | <>   Not equal
   | !=   Not equal
   | <    Less than
   | >    Greater than
   | <=   Less than or equal to
   | >=   Greater than or equal to

quantifier is:
   ALL | ANY | SOME
```

row-value-constructor

specifies the first operand of a quantified comparison predicate. The first operand can be either of:

(*expression* [,*expression*]...)

is a sequence of SQL value expressions, separated by commas and enclosed in parentheses. *expression* cannot include an aggregate function unless *expression* is in a HAVING clause. *expression* can be a scalar subquery (a subquery that returns a single row consisting of a single column). See "Expressions" (page 211).

row-subquery

is a subquery that returns a single row (consisting of a sequence of values). See "Subquery" (page 252).

ALL

specifies that the predicate is true if the comparison is true for every row selected by *table-subquery* (or if *table-subquery* selects no rows), and specifies that the predicate is false if the comparison is false for at least one row selected.

ANY | SOME

specifies that the predicate is true if the comparison is true for at least one row selected by the *table-subquery* and specifies that the predicate is false if the comparison is false for every row selected (or if *table-subquery* selects no rows).

table-subquery

provides the values for the comparison. The number of values returned by the *row-value-constructor* must be equal to the number of values specified by the

table-subquery, and the data types of values returned by the *row-value-constructor* must be comparable to the data types of values returned by the *table-subquery*. See "Subquery" (page 252).

Considerations for ALL, ANY, SOME

Let *R* be the result of the *row-value-constructor*, *T* the result of the *table-subquery*, and *RT* a row in *T*.

Result of *Rcomparison-op* **ALL** *T*

If *T* is empty or if *R comparison-op RT* is true for every row *RT* in *T*, the *comparison-op* ALL predicate is true.

If *R comparison-op RT* is false for at least one row *RT* in *T*, the *comparison-op* ALL predicate is false.

Result of *Rcomparison-op* **ANY** *T* **or** *Rcomparison-op* **SOME** *T*

If *T* is empty or if *R comparison-op RT* is false for every row *RT* in *T*, the *comparison-op* ANY predicate is false.

If *R comparison-op RT* is true for at least one row *RT* in *T*, the *comparison-op* ANY predicate is true.

Examples of ALL, ANY, SOME

- This predicate is true if the salary is greater than the salaries of all the employees who have a jobcode of 420:

```
salary > ALL (SELECT salary
              FROM persnl.employee
              WHERE jobcode = 420)
```

Consider this SELECT statement using the preceding predicate:

```
SELECT empnum, first_name, last_name, salary
FROM persnl.employee
WHERE salary > ALL (SELECT salary
                    FROM persnl.employee
                    WHERE jobcode = 420);
```

The inner query providing the comparison values yields these results:

```
SELECT salary
FROM persnl.employee
WHERE jobcode = 420;

SALARY
-----------
   33000.00
   36000.00
   18000.10

--- 3 row(s) selected.
```

The SELECT statement using this inner query yields these results. The salaries listed are greater than the salary of every employees with jobcode equal to 420—that is, greater than $33,000.00, $36,000.00, and $18,000.10:

```
SELECT empnum, first_name, last_name, salary
FROM persnl.employee
WHERE salary > ALL (SELECT salary
                    FROM persnl.employee
                    WHERE jobcode = 420);

EMPNUM  FIRST_NAME       LAST_NAME            SALARY
------  ---------------  -------------------  -----------
     1  ROGER            GREEN                  175500.00
    23  JERRY            HOWARD                 137000.10
```

```
    29  JANE             RAYMOND                 136000.00
   ...
   343  ALAN             TERRY                    39500.00
   557  BEN              HENDERSON                65000.00
   568  JESSICA          CRINER                   39500.00

--- 23 row(s) selected.
```

- This predicate is true if the part number is equal to any part number with more than five units in stock:

```
partnum = ANY (SELECT partnum
               FROM sales.odetail
               WHERE qty_ordered > 5)
```

Consider this SELECT statement using the preceding predicate:

```
SELECT ordernum, partnum, qty_ordered
FROM sales.odetail
WHERE partnum = ANY (SELECT partnum
                     FROM sales.odetail
                    WHERE qty_ordered > 5);
```

The inner query providing the comparison values yields these results:

```
SELECT partnum
FROM sales.odetail
WHERE qty_ordered > 5;

Part/Num
--------
    2403
    5100
    5103
    6301
    6500
    ....

--- 60 row(s) selected.
```

The SELECT statement using this inner query yields these results. All of the order numbers listed have part number equal to any part number with more than five total units in stock—that is, equal to 2403, 5100, 5103, 6301, 6500, and so on:

```
SELECT ordernum, partnum, qty_ordered
FROM sales.odetail
WHERE partnum = ANY (SELECT partnum
                     FROM sales.odetail
                     WHERE qty_ordered > 5);

Order/Num   Part/Num  Qty/Ord
----------  --------  ----------
    100210       244           3
    100210      2001           3
    100210      2403           6
    100210      5100          10
    100250       244           4
    100250      5103          10
    100250      6301          15
    100250      6500          10
    ......      ....          ..

--- 71 row(s) selected.
```

Privileges

A privilege provides authorization to perform a specific operation for a specific object.

A privilege can be granted to or revoked from a user or role in many ways:

- Implicit privileges are granted to an owner of an object when the object is created. The owner retains implicit privileges for the lifespan of the object.
- Explicit privileges can be granted to or revoked from a user or role. Explicit privileges can be granted or revoked by a database user administrator, an object owner, or a user who has been granted the privilege with the WITH GRANT OPTION option.
- The privileges granted to a user can come from various sources. Privileges can be directly granted to a user or they can be inherited through a role. For example, a user gets the SELECT privilege on table T1 from two different roles. If one of the roles is revoked from the user, the user will still be able to select from T1 via the SELECT privilege granted to the remaining role.
- A user who is granted a role is thereby conferred all privileges of the role. The only way to revoke any such privilege is to revoke the role from the user. For more information, see "Roles" (page 248).

You can manage privileges by using the GRANT and REVOKE statements. For more information on GRANT, see:

- "GRANT Statement" (page 111)
- "GRANT COMPONENT PRIVILEGE Statement" (page 114)
- "GRANT ROLE Statement" (page 117)

For more information on REVOKE, see:

- "REVOKE Statement" (page 130)
- "REVOKE COMPONENT PRIVILEGE Statement" (page 133)
- "REVOKE ROLE Statement" (page 135)

Roles

A role offers the flexibility of implicitly assigning a set of privileges to users, instead of assigning privileges individually. A user can be granted one or more roles. A role can be granted to one or more users. A role can be granted by or revoked by a database user administrator, a role owner, or a member of the role.

Privileges are granted to a role. When a role is granted to a user, the privileges granted to the role become available to the user. If new privileges are granted to the role, those privileges become available to all users who have been granted the role. When a role is revoked from a user, the privileges granted to the role are no longer available to the user. In Trafodion Release 0.9, for any privilege changes to take effect, the user must disconnect any current sessions and then reconnect to establish new sessions. However, starting in Trafodion Release 1.0, the change in privileges is automatically propagated to and detected by active sessions, so there is no need for users to disconnect from and reconnect to a session to see the updated set of privileges. For more information about privileges, see "Privileges" (page 247).

A role name is an authorization ID. A role name cannot be identical to a registered database username. For more information, see "Authorization IDs" (page 193).

To manage roles, see these SQL statements:

- "CREATE ROLE Statement" (page 66)
- "DROP ROLE Statement" (page 93)
- "GRANT ROLE Statement" (page 117)
- "REVOKE ROLE Statement" (page 135)

Schemas

The ANSI SQL:1999 schema name is an SQL identifier that is unique for a given ANSI catalog name. Trafodion SQL automatically qualifies the schema name with the current default catalog name, TRAFODION.

The logical name of the form *schema.object* is an ANSI name. The part *schema* denotes the ANSI-defined schema.

To be compliant with ANSI SQL:1999, Trafodion SQL provides support for ANSI object names. By using these names, you can develop ANSI-compliant applications that access all SQL objects. You can access Trafodion SQL objects with the name of the actual object. See "SET SCHEMA Statement" (page 156).

Creating and Dropping Schemas

In Trafodion Release 0.9 and earlier, a schema is created when you qualify a table or view name with a new schema name in a CREATE TABLE or CREATE VIEW statement. A schema is dropped when all the database objects in the schema have been dropped. The CREATE SCHEMA statement would run but not do anything.

Starting in Trafodion Release 1.0.0, you can now create a schema using the CREATE SCHEMA command and drop a schema using the DROP SCHEMA statement. For more information, see the "CREATE SCHEMA Statement" (page 67) and the "DROP SCHEMA Statement" (page 95).

Search Condition

A search condition is used to choose rows from tables or views, depending on the result of applying the condition to rows. The condition is a Boolean expression consisting of predicates combined together with OR, AND, and NOT operators.

You can use a search condition in the WHERE clause of a SELECT, DELETE, or UPDATE statement, the HAVING clause of a SELECT statement, the searched form of a CASE expression, the ON clause of a SELECT statement that involves a join, a CHECK constraint, or a ROWS SINCE sequence function.

```
search-condition is:
   boolean-term | search-condition OR boolean-term

boolean-term is:
   boolean-factor | boolean-term AND boolean-factor

boolean-factor is:
   [NOT] boolean-primary

boolean-primary is:
   predicate | (search-condition)
```

OR

specifies the resulting search condition is true if and only if either of the surrounding predicates or search conditions is true.

AND

specifies the resulting search condition is true if and only if both the surrounding predicates or search conditions are true.

NOT

reverses the truth value of its operand—the following predicate or search condition.

predicate

is a BETWEEN, comparison, EXISTS, IN, LIKE, NULL, or quantified comparison predicate. A predicate specifies conditions that must be satisfied for a row to be chosen. See "Predicates" (page 233) and individual entries.

Considerations for Search Condition

Order of Evaluation

SQL evaluates search conditions in this order:

1. Predicates within parentheses
2. NOT
3. AND
4. OR

Column References

Within a search condition, a reference to a column refers to the value of that column in the row currently being evaluated by the search condition.

Subqueries

If a search condition includes a subquery and the subquery returns no values, the predicate evaluates to null. See "Subquery" (page 252).

Examples of Search Condition

- Select rows by using a search condition composed of three comparison predicates joined by AND operators:

```
select O.ordernum, O.deliv_date, OD.qty_ordered
FROM sales.orders O,
  sales.odetail OD
WHERE qty_ordered < 9 AND deliv_date <= DATE '2008-11-01'
  AND O.ordernum = OD.ordernum;

ORDERNUM    DELIV_DATE  QTY_ORDERED
----------  ----------  -----------
    100210  2008-04-10            3
    100210  2008-04-10            3
    100210  2008-04-10            6
    100250  2008-06-15            4
    101220  2008-12-15            3
       ...
--- 28 row(s) selected.
```

- Select rows by using a search condition composed of three comparison predicates, two of which are joined by an OR operator (within parentheses), and where the result of the OR and the first comparison predicate are joined by an AND operator:

```
SELECT partnum, S.suppnum, suppname
FROM invent.supplier S,
  invent.partsupp PS
WHERE S.suppnum = PS.suppnum
  AND (partnum < 3000 OR partnum = 7102);

PARTNUM  SUPPNUM  SUPPNAME
-------  -------  ------------------
    212        1  NEW COMPUTERS INC
    244        1  NEW COMPUTERS INC
    255        1  NEW COMPUTERS INC
    ...
   7102       10  LEVERAGE INC

--- 18 row(s) selected.
```

Subquery

A subquery is a query expression enclosed in parentheses. Its syntactic form is specified in the syntax of a SELECT statement. For further information about query expressions, see "SELECT Statement" (page 138).

A subquery is used to provide values for a BETWEEN, comparison, EXISTS, IN, or quantified comparison predicate in a search condition. It is also used to specify a derived table in the FROM clause of a SELECT statement.

A subquery can be a table, row, or scalar subquery. Therefore, its result table can be a table consisting of multiple rows and columns, a single row of column values, or a single row consisting of only one column value.

SELECT Form of a Subquery

A subquery is typically specified as a special form of a SELECT statement enclosed in parentheses that queries (or selects) to provide values in a search condition or to specify a derived table as a table reference.

The form of a subquery specified as a SELECT statement is *query-expr*.

Neither the ORDER BY clause nor [FIRST N] / [ANY N] clause is allowed in a subquery.

Using Subqueries to Provide Comparison Values

When a subquery is used to provide comparison values, the SELECT statement that contains the subquery is called an outer query. The subquery within the SELECT is called an *inner query*. In this case, the differences between the SELECT statement and the SELECT form of a subquery are:

- A subquery is always enclosed in parentheses.
- A subquery cannot contain an ORDER BY clause.
- If a subquery is not part of an EXISTS, IN, or quantified comparison predicate, and the subquery evaluates to more than one row, a run-time error occurs.

Nested Subqueries When Providing Comparison Values

An outer query (a main SELECT statement) can have nested subqueries. Subqueries within the same WHERE or HAVING clause are at the same level. For example, this query has one level of nesting:

```
SELECT * FROM TABLE1
  WHERE A = (SELECT P FROM TABLE2 WHERE Q = 1)
    AND B = (SELECT X FROM TABLE3 WHERE Y = 2)
```

A subquery within the WHERE clause of another subquery is at a different level, however, so this query has two levels of nesting:

```
SELECT * FROM TABLE1
  WHERE A = (SELECT P FROM TABLE2
             WHERE Q = (SELECT X FROM TABLE3
                        WHERE Y = 2))
```

The maximum level of nested subqueries might depend on:

- The complexity of the subqueries.
- Whether the subquery is correlated and if so, whether it can be unnested.
- Amount of available memory.

Other factors may affect the maximum level of subqueries.

Correlated Subqueries When Providing Comparison Values

In a subquery, when you refer to columns of any table or view defined in an outer query, the reference is called an outer reference. A subquery containing an outer reference is called a correlated subquery.

If you refer to a column name that occurs in more than one outer query, you must qualify the column name with the correlation name of the table or view to which it belongs. Similarly, if you refer to a column name that occurs in the subquery and in one or more outer queries, you must qualify the column name with the correlation name of the table or view to which it belongs. The correlation name is known to other subqueries at the same level, or to inner queries but not to outer queries.

If you use the same correlation name at different levels of nesting, an inner query uses the one from the nearest outer level.

Tables

A table is a logical representation of data in which a set of records is represented as a sequence of rows, and the set of fields common to all rows is represented by columns. A column is a set of values of the same data type with the same definition. The intersection of a row and column represents the data value of a particular field in a particular record.

Every table must have one or more columns, but the number of rows can be zero. No inherent order of rows exists within a table.

You create a Trafodion SQL user table by using the CREATE TABLE statement. See the "CREATE TABLE Statement" (page 69). The definition of a user table within the statement includes this information:

- Name of the table
- Name of each column of the table
- Type of data you can store in each column of the table
- Other information about the table, including the physical characteristics of the file that stores the table (for example, the storage order of rows within the table)

A Trafodion SQL table is described in an SQL schema and stored as an HBase table. Trafodion SQL tables have regular ANSI names in the catalog TRAFODION. A Trafodion SQL table name can be a fully qualified ANSI name of the form `TRAFODION.`*`schema-name`*`.`*`object-name`*. A Trafodion SQL table's metadata is stored in the schema `TRAFODION."_MD_"`.

Because Trafodion defines the encodings for column values in Trafodion SQL tables, those tables support various Trafodion SQL statements. See "Supported SQL Statements With HBase Tables" (page 23).

Internally, Trafodion SQL tables use a single HBase column family and shortened column names to conserve space. Their encoding allows keys consisting of multiple columns and preserves the order of key values as defined by SQL. The underlying HBase column model makes it very easy to add and remove columns from Trafodion SQL tables. HBase columns that are not recorded in the Trafodion metadata are ignored, and missing columns are considered NULL values.

Base Tables and Views

In some descriptions of SQL, tables created with a CREATE TABLE statement are called base tables to distinguish them from views, which are called logical tables.

A view is a named logical table defined by a query specification that uses one or more base tables or other views. See "Views" (page 255).

Example of a Base Table

For example, this EMPLOYEE table is a base table in a sample database:

EMPNUM	FIRST_NAME	LAST_NAME	DEPTNUM	JOBCODE	SALARY
1	ROGER	GREEN	9000	100	175500.00
23	JERRY	HOWARD	1000	100	137000.00
75	TIM	WALKER	3000	300	32000.00
. . .	. . .	. . .	. . .	. . .	. . .

In this sample table, the columns are EMPNUM, FIRST_NAME, LAST_NAME, DEPTNUM, JOBCODE, and SALARY. The values in each column have the same data type.

Views

A view provides an alternate way of looking at data in one or more tables. A view is a named specification of a result table, which is a set of rows selected or generated from one or more base tables or other views. The specification is a SELECT statement that is executed whenever the view is referenced.

A view is a logical table created with the CREATE VIEW statement and derived by projecting a subset of columns, restricting a subset of rows, or both, from one or more base tables or other views.

SQL Views

A view's name must be unique among table and view names within the schema that contains it.

Single table views can be updatable. Multitable views are not updatable.

For information about SQL views, see "CREATE VIEW Statement" (page 81) and "DROP VIEW Statement" (page 97).

Example of a View

You can define a view to show only part of the data in a table. For example, this EMPLIST view is defined as part of the EMPLOYEE table:

EMPNUM	FIRST_NAME	LAST_NAME	DEPTNUM	JOBCODE
1	ROGER	GREEN	9000	100
23	JERRY	HOWARD	1000	100
75	TIM	WALKER	3000	300
. . .	. . .	. . .	. . .	. . .

In this sample view, the columns are EMPNUM, FIRST_NAME, LAST_NAME, DEPTNUM, and JOBCODE. The SALARY column in the EMPLOYEE table is not part of the EMPLIST view.

5 SQL Clauses

Clauses are used by Trafodion SQL statements to specify default values, ways to sample or sort data, how to store physical data, and other details.

This section describes:

- "DEFAULT Clause" specifies a default value for a column being created.
- "FORMAT Clause" specifies the format to use.
- "SAMPLE Clause" specifies the sampling method used to select a subset of the intermediate result table of a SELECT statement.
- "SEQUENCE BY Clause" specifies the order in which to sort rows of the intermediate result table for calculating sequence functions.
- "TRANSPOSE Clause" generates, for each row of the SELECT source table, a row for each item in the transpose item list.

DEFAULT Clause

"Examples of DEFAULT"

The DEFAULT option of the CREATE TABLE or ALTER TABLE *table-name* ADD COLUMN statement specifies a default value for a column being created. The default value is used when a row is inserted in the table without a value for the column.

```
DEFAULT default | NO DEFAULT

 default is:
     literal
   | NULL
   | CURRENT_DATE
   | CURRENT_TIME
   | CURRENT_TIMESTAMP
```

NO DEFAULT

specifies the column has no default value. You cannot specify NO DEFAULT in an ALTER TABLE statement. See "ALTER TABLE Statement" (page 36).

DEFAULT *literal*

is a literal of a data type compatible with the data type of the associated column.

For a character column, *literal* must be a string literal of no more than 240 characters or the length of the column, whichever is less. The maximum length of a default value for a character column is 240 bytes (minus control characters) or the length of the column, whichever is less. Control characters consist of character set prefixes and single quote delimiter found in the text itself.

For a numeric column, *literal* must be a numeric literal that does not exceed the defined length of the column. The number of digits to the right of the decimal point must not exceed the scale of the column, and the number of digits to the left of the decimal point must not exceed the number in the length (or length minus scale, if you specified scale for the column).

For a datetime column, *literal* must be a datetime literal with a precision that matches the precision of the column.

For an INTERVAL column, *literal* must be an INTERVAL literal that has the range of INTERVAL fields defined for the column.

DEFAULT NULL

specifies NULL as the default. This default can occur only with a column that allows null.

DEFAULT CURRENT_DATE

specifies the default value for the column as the value returned by the CURRENT_DATE function at the time of the operation that assigns a value to the column. This default can occur only with a column whose data type is DATE.

DEFAULT CURRENT_TIME

specifies the default value for the column as the value returned by the CURRENT_TIME function at the time of the operation that assigns a value to the column. This default can occur only with a column whose data type is TIME.

DEFAULT CURRENT_TIMESTAMP

specifies the default value for the column as the value returned by the CURRENT_TIMESTAMP function at the time of the operation that assigns a value to the column. This default can occur only with a column whose data type is TIMESTAMP.

Examples of DEFAULT

- This example uses DEFAULT clauses on CREATE TABLE to specify default column values:

```
CREATE TABLE items
  ( item_id      CHAR(12)    NO DEFAULT
    ,description CHAR(50)    DEFAULT NULL
    ,num_on_hand INTEGER     DEFAULT 0 NOT NULL );
```

- This example uses DEFAULT clauses on CREATE TABLE to specify default column values:

```
CREATE TABLE persnl.project
  ( projcode           NUMERIC (4) UNSIGNED
                       NO DEFAULT
                       NOT NULL
    ,empnum            NUMERIC (4) UNSIGNED
                       NO DEFAULT
                       NOT NULL
    ,projdesc          VARCHAR (18)
                       DEFAULT NULL
    ,start_date        DATE
                       DEFAULT CURRENT_DATE
    ,ship_timestamp    TIMESTAMP
                       DEFAULT CURRENT_TIMESTAMP
    ,est_complete      INTERVAL DAY
                       DEFAULT INTERVAL '30' DAY
    ,PRIMARY KEY      (projcode) );
```

FORMAT Clause

- "Considerations for Date Formats"
- "Considerations for Other Formats"
- "Examples of FORMAT"

The FORMAT clause specifies the output format for DATE values. It can also be used to specify the length of character output or to specify separating the digits of integer output with colons.

Date Formats:

```
(FORMAT 'format-string') |

(DATE, FORMAT 'format-string')

format-string for Date Formats is:
   YYYY-MM-DD
   MM/DD/YYYY
   YY/MM/DD
   YYYY/MM/DD
   YYYYMMDD
   DD.MM.YYYY
   DD-MM-YYYY
   DD-MMM-YYYY
```

Other Formats:

```
(FORMAT 'format-string')

format-string for other formats is:
   XXX
   99:99:99:99
   -99:99:99:99
```

YYYY-MM-DD
: specifies that the FORMAT clause output format is *year-month-day*.

MM/DD/YYYY
: specifies that the FORMAT clause output format is *month/day/year*

YY/MM/DD
: specifies that the FORMAT clause output format is *year/month/day*.

YYYY/MM/DD
: specifies that the FORMAT clause output format is *year/month/day*.

YYYYMMDD
: specifies that the FORMAT clause output format is *yearmonthday*.

DD.MM.YYYY
: specifies that the FORMAT clause output format is *day.month.year*.

DD-MM-YYYY
: specifies that the FORMAT clause output format is *day-month-year*.

DD-MMM-YYYY
: specifies that the FORMAT clause output format is *day-month-year*.

XXX
: specifies that the FORMAT clause output format is a string format. The input must be a numeric or string value.

99:99:99:99
: specifies that the FORMAT clause output format is a timestamp. The input must be a numeric value.

-99:99:99:99

specifies that the FORMAT clause output format is a timestamp. The input must be a numeric value.

Considerations for Date Formats

The expression preceding the (FORMAT "*format-string*') clause must be a DATE value.

The expression preceding the (DATE, FORMAT '*format-string*') clause must be a quoted string in the USA, EUROPEAN, or DEFAULT date format.

Considerations for Other Formats

For XXX, the expression preceding the (FORMAT '*format-string*') clause must be a numeric value or a string value.

For 99:99:99:99 and -99:99:99:99, the expression preceding the (FORMAT '*format-string*') clause must be a numeric value.

Examples of FORMAT

The format string 'XXX' in this example will yield a sample result of `abc`:

```
SELECT 'abcde' (FORMAT 'XXX') FROM (VALUES(1)) t;
```

The format string 'YYYY-MM_DD' in this example will yield a sample result of `2008-07-17`.

```
SELECT CAST('2008-07-17' AS DATE) (FORMAT 'YYYY-MM-DD') FROM (VALUES(1)) t;
```

The format string 'MM/DD/YYYY' in this example will yield a sample result of `07/17/2008`.

```
SELECT '2008-07-17' (DATE, FORMAT 'MM/DD/YYYY') FROM (VALUES(1)) t;
```

The format string 'YY/MM/DD' in this example will yield a sample result of `08/07/17`.

```
SELECT '2008-07-17'(DATE, FORMAT 'YY/MM/DD') FROM (VALUES(1)) t;
```

The format string 'YYYY/MM/DD' in this example will yield a sample result of `2008/07/17`.

```
SELECT '2008-07-17' (DATE, FORMAT 'YYYY/MM/DD') FROM (VALUES(1)) t;
```

The format string 'YYYYMMDD' in this example will yield a sample result of `20080717`.

```
SELECT '2008-07-17' (DATE, FORMAT 'YYYYMMDD') FROM (VALUES(1)) t;
```

The format string 'DD.MM.YYYY' in this example will yield a sample result of `17.07.2008`.

```
SELECT '2008-07-17' (DATE, FORMAT 'DD.MM.YYYY') FROM (VALUES(1)) t;
```

The format string 'DD-MMM-YYYY' in this example will yield a sample result of `17-JUL-2008`.

```
SELECT '2008-07-17' (DATE, FORMAT 'DD-MMM-YYYY') FROM (VALUES(1)) t;
```

The format string '99:99:99:99' in this example will yield a sample result of `12:34:56:78`.

```
SELECT 12345678 (FORMAT '99:99:99:99') FROM (VALUES(1)) t;
```

The format string '-99:99:99:99' in this example will yield a sample result of `-12:34:56:78`.

```
SELECT (-12345678) (FORMAT '-99:99:99:99') FROM (VALUES(1)) t;
```

SAMPLE Clause

- "Considerations for SAMPLE"
- "Examples of SAMPLE"

The SAMPLE clause of the SELECT statement specifies the sampling method used to select a subset of the intermediate result table of a SELECT statement. The intermediate result table consists of the rows returned by a WHERE clause or, if no WHERE clause exists, the FROM clause. See "SELECT Statement" (page 138).

SAMPLE is a Trafodion SQL extension.

```
SAMPLE sampling-method

sampling-method is:
     RANDOM percent-size
   | FIRST rows-size
              [SORT BY colname [ASC[ENDING] | DESC[ENDING]]
                [,colname [ASC[ENDING] | DESC[ENDING]]]...]
   | PERIODIC rows-size EVERY number-rows ROWS
              [SORT BY colname [ASC[ENDING] | DESC[ENDING]]
                [,colname [ASC[ENDING] | DESC[ENDING]]]...]

percent-size is:
     percent-result PERCENT [ROWS]
   | BALANCE WHEN condition
       THEN percent-result PERCENT [ROWS]
       [WHEN condition THEN percent-result PERCENT [ROWS]]...
       [ELSE percent-result PERCENT [ROWS]] END

rows-size is:
     number-rows ROWS
   | BALANCE WHEN condition THEN number-rows ROWS
            [WHEN condition THEN number-rows ROWS]...
            [ELSE number-rows ROWS] END
```

RANDOM *percent-size*

directs Trafodion SQL to choose rows randomly (each row having an unbiased probability of being chosen) without replacement from the result table. The sampling size is determined by the *percent-size*, defined as:

percent-result PERCENT [ROWS] | BALANCE WHEN *condition* THEN *percent-result* PERCENT [ROWS] [WHEN *condition* THEN *percent-result* PERCENT [ROWS]]... [ELSE *percent-result* PERCENT [ROWS]] END

specifies the value of the size for RANDOM sampling by using a percent of the result table. The value *percent-result* must be a numeric literal.

You can determine the actual size of the sample. Suppose that *N* rows exist in the intermediate result table. Each row is picked with a probability of *r*%, where *r* is the sample size in PERCENT. Therefore, the actual size of the resulting sample is approximately *r*% of *N*. The number of rows picked follows a binomial distribution with mean equal to *r* * *N*/100.

If you specify a sample size greater than 100 PERCENT, Trafodion SQL returns all the rows in the result table plus duplicate rows. The duplicate rows are picked from the result table according to the specified sampling method. This technique is called oversampling.

ROWS

specifies row sampling. Row sampling is the default.

BALANCE

If you specify a BALANCE expression, Trafodion SQL performs stratified sampling. The intermediate result table is divided into disjoint strata based on the WHEN conditions.

Each stratum is sampled independently by using the sampling size. For a given row, the stratum to which it belongs is determined by the first WHEN condition that is true for that row—if a true condition exists. If no true condition exists, the row belongs to the ELSE stratum.

FIRST *rows-size* [SORT BY *colname* [ASC[ENDING] | DESC[ENDING]] [,*colname* [ASC[ENDING] | DESC[ENDING]]]...]

directs Trafodion SQL to choose the first rows from the result table. You can specify the order of the rows to sample. Otherwise, Trafodion SQL chooses an arbitrary order. The sampling size is determined by the *rows-size*, defined as:

number-rows ROWS | BALANCE WHEN *condition* THEN *number-rows* ROWS [WHEN *condition* THEN *number-rows* ROWS]... [ELSE *number-rows* ROWS] END

specifies the value of the size for FIRST sampling by using the number of rows intended in the sample. The value *number-rows* must be an integer literal.

You can determine the actual size of the sample. Suppose that *N* rows exist in the intermediate result table. If the size *s* of the sample is specified as a number of rows, the actual size of the resulting sample is the minimum of *s* and *N*.

PERIODIC *rows-size* EVERY *number-rows* ROWS [SORT BY *colname* [ASC[ENDING] | DESC[ENDING]] [,*colname* [ASC[ENDING] | DESC[ENDING]]]...]

directs Trafodion SQL to choose the first rows from each block (or period) of contiguous rows. This sampling method is equivalent to a separate FIRST sampling for each period, and the *rows-size* is defined as in FIRST sampling.

The size of the period is specified as a number of rows. You can specify the order of the rows to sample. Otherwise, Trafodion SQL chooses an arbitrary order.

You can determine the actual size of the sample. Suppose that *N* rows exist in the intermediate result table. If the size *s* of the sample is specified as a number of rows and the size *p* of the period is specified as a number of rows, the actual size of the resulting sample is calculated as:

FLOOR (N/p) * s + *minimum* (MOD (N, p), s)

minimum in this expression is used simply as the mathematical minimum of two values.

Considerations for SAMPLE

Sample Rows

In general, when you use the SAMPLE clause, the same query returns different sets of rows for each execution. The same set of rows is returned only when you use the FIRST and PERIODIC sampling methods with the SORT BY option, where no duplicates exist in the specified column combination for the sort.

Examples of SAMPLE

Suppose that the data-mining tables SALESPER, SALES, and DEPT have been created as:

```
CREATE TABLE trafodion.mining.salesper
   ( empid   NUMERIC (4) UNSIGNED NOT NULL
    ,dnum    NUMERIC (4) UNSIGNED NOT NULL
    ,salary  NUMERIC (8,2) UNSIGNED
    ,age     INTEGER
    ,sex     CHAR (6)
    ,PRIMARY KEY (empid) );

CREATE TABLE trafodion.mining.sales
    ( empid   NUMERIC (4) UNSIGNED NOT NULL
     ,product VARCHAR (20)
     ,region  CHAR (4)
```

```
     ,amount  NUMERIC (9,2) UNSIGNED
     ,PRIMARY KEY (empid) );

CREATE TABLE trafodion.mining.dept
    ( dnum    NUMERIC (4) UNSIGNED NOT NULL
     ,name    VARCHAR (20)
     ,PRIMARY KEY (dnum) );
```

Suppose, too, that sample data is inserted into this database.

- Return the SALARY of the youngest 50 sales people:

```
SELECT salary
FROM salesperson
SAMPLE FIRST 50 ROWS SORT BY age;

SALARY
-----------
   90000.00
   90000.00
   28000.00
   27000.12
  136000.00
   37000.40
        ...

--- 50 row(s) selected.
```

- Return the SALARY of 50 sales people. In this case, the table is clustered on EMPID. If the optimizer chooses a plan to access rows using the primary access path, the result consists of salaries of the 50 sales people with the smallest employee identifiers.

```
SELECT salary
FROM salesperson
SAMPLE FIRST 50 ROWS;

SALARY
-----------
  175500.00
  137000.10
  136000.00
  138000.40
   75000.00
   90000.00
        ...

--- 50 row(s) selected.
```

- Return the SALARY of the youngest five sales people, skip the next 15 rows, and repeat this process until no more rows exist in the intermediate result table. You cannot specify periodic sampling with the sample size larger than the period.

```
SELECT salary
FROM salesperson
SAMPLE PERIODIC 5 ROWS EVERY 20 ROWS SORT BY age;

SALARY
-----------
   90000.00
   90000.00
   28000.00
   27000.12
  136000.00
   36000.00
```

```
        ...

--- 17 row(s) selected.
```

In this example, 62 rows exist in the SALESPERSON table. For each set of 20 rows, the first five rows are selected. The last set consists of two rows, both of which are selected.

- Compute the average salary of a random 10 percent of the sales people. You will get a different result each time you run this query because it is based on a random sample.

```
SELECT AVG(salary)
FROM salesperson
SAMPLE RANDOM 10 PERCENT;

(EXPR)
--------------------
            61928.57

--- 1 row(s) selected.
```

- This query illustrates sampling after execution of the WHERE clause has chosen the qualifying rows. The query computes the average salary of a random 10 percent of the sales people over 35 years of age. You will get a different result each time you run this query because it is based on a random sample.

```
SELECT AVG(salary)
FROM salesperson
WHERE age > 35
SAMPLE RANDOM 10 PERCENT;

(EXPR)
--------------------
            58000.00

--- 1 row(s) selected.
```

- Compute the average salary of a random 10 percent of sales people belonging to the CORPORATE department. The sample is taken from the join of the SALESPERSON and DEPARTMENT tables. You will get a different result each time you run this query because it is based on a random sample.

```
SELECT AVG(salary)
FROM salesperson S, department D
WHERE S.DNUM = D.DNUM
AND D.NAME = 'CORPORATE'
SAMPLE RANDOM 10 PERCENT;

(EXPR)
---------------------
           106250.000

--- 1 row(s) selected.
```

- In this example, the SALESPERSON table is first sampled and then joined with the DEPARTMENT table. This query computes the average salary of all the sales people belonging to the CORPORATE department in a random sample of 10 percent of the sales employees.

```
SELECT AVG(salary)
FROM ( SELECT salary, dnum
       FROM salesperson
       SAMPLE RANDOM 10 PERCENT ) AS S, department D
WHERE S.DNUM = D.DNUM
AND D.NAME = 'CORPORATE';
```

```
(EXPR)
--------------------
           37000.000

--- 1 row(s) selected.
```

The results of this query and some of the results of previous queries might return null:

```
SELECT AVG(salary)
FROM ( SELECT salary, dnum
       FROM salesperson
       SAMPLE RANDOM 10 PERCENT ) AS S, department D
WHERE S.DNUM = D.DNUM
AND D.NAME = 'CORPORATE';

(EXPR)
--------------------
                   ?

--- 1 row(s) selected.
```

For this query execution, the number of rows returned by the embedded query is limited by the total number of rows in the SALESPERSON table. Therefore, it is possible that no rows satisfy the search condition in the WHERE clause.

- In this example, both the tables are sampled first and then joined. This query computes the average salary and the average sale amount generated from a random 10 percent of all the sales people and 20 percent of all the sales transactions.

```
SELECT AVG(salary), AVG(amount)
FROM ( SELECT salary, empid
       FROM salesperson
       SAMPLE RANDOM 10 PERCENT ) AS S,
     ( SELECT amount, empid
       FROM sales
       SAMPLE RANDOM 20 PERCENT ) AS T
WHERE S.empid = T.empid;

(EXPR)      (EXPR)
---------   ---------
 45000.00    31000.00

--- 1 row(s) selected.
```

- This example illustrates oversampling. This query retrieves 150 percent of the sales transactions where the amount exceeds $1000. The result contains every row at least once, and 50 percent of the rows, picked randomly, occur twice.

```
SELECT *
FROM sales
WHERE amount > 1000
SAMPLE RANDOM 150 PERCENT;

EMPID  PRODUCT               REGION  AMOUNT
-----  --------------------  ------  -----------
    1  PCGOLD, 30MB          E          30000.00
   23  PCDIAMOND, 60MB       W          40000.00
   23  PCDIAMOND, 60MB       W          40000.00
   29  GRAPHICPRINTER, M1    N          11000.00
   32  GRAPHICPRINTER, M2    S          15000.00
   32  GRAPHICPRINTER, M2    S          15000.00
  ...  ...                  ...              ...
```

```
--- 88 row(s) selected.
```

- The BALANCE option enables stratified sampling. Retrieve the age and salary of 1000 sales people such that 50 percent of the result are male and 50 percent female.

```
SELECT age, sex, salary
FROM salesperson
SAMPLE FIRST
       BALANCE WHEN sex = 'male' THEN 15 ROWS
               WHEN sex = 'female' THEN 15 ROWS
       END
ORDER BY age;

AGE             SEX      SALARY
-----------  ------  -----------
         22  male       28000.00
         22  male       90000.00
         22  female    136000.00
         22  male       37000.40
          ...  ...             ...

--- 30 row(s) selected.
```

- Retrieve all sales records with the amount exceeding $10000 and a random sample of 10 percent of the remaining records:

```
SELECT *
FROM sales
SAMPLE RANDOM
       BALANCE WHEN amount > 10000 THEN 100 PERCENT
       ELSE 10 PERCENT
       END;

EMPID  PRODUCT               REGION  AMOUNT
-----  --------------------  ------  -----------
    1  PCGOLD, 30MB          E          30000.00
   23  PCDIAMOND, 60MB       W          40000.00
   29  GRAPHICPRINTER, M1    N          11000.00
   32  GRAPHICPRINTER, M2    S          15000.00
  ...  ...                   ...             ...
  228  MONITORCOLOR, M2      N          10500.00
  ...  ...                   ...             ...

--- 32 row(s) selected.
```

- This query shows an example of stratified sampling where the conditions are not mutually exclusive:

```
SELECT *
FROM sales
SAMPLE RANDOM
BALANCE WHEN amount > 10000 THEN 100 PERCENT
        WHEN product = 'PCGOLD, 30MB' THEN 25 PERCENT
        WHEN region = 'W' THEN 40 PERCENT
        ELSE 10 PERCENT
        END;

EMPID  PRODUCT               REGION  AMOUNT
-----  --------------------  ------  -----------
    1  PCGOLD, 30MB          E          30000.00
   23  PCDIAMOND, 60MB       W          40000.00
   29  GRAPHICPRINTER, M1    N          11000.00
   32  GRAPHICPRINTER, M2    S          15000.00
```

```
   39  GRAPHICPRINTER, M3    S          20000.00
   75  LASERPRINTER, X1      W          42000.00
  ...  ...                   ...             ...

--- 30 row(s) selected.
```

SEQUENCE BY Clause

- "Considerations for SEQUENCE BY"
- "Examples of SEQUENCE BY"

The SEQUENCE BY clause of the SELECT statement specifies the order in which to sort the rows of the intermediate result table for calculating sequence functions. This option is used for processing time-sequenced rows in data mining applications. See "SELECT Statement" (page 138).

SEQUENCE BY is a Trafodion SQL extension.

```
SEQUENCE BY colname [ASC[ENDING] | DESC[ENDING]]
   [,colname [ASC[ENDING] | DESC[ENDING]]]...
```

colname

names a column in *select-list* or a column in a table reference in the FROM clause of the SELECT statement. *colname* is optionally qualified by a table, view, or correlation name; for example, CUSTOMER.CITY.

ASC | DESC

specifies the sort order. ASC is the default. For ordering an intermediate result table on a column that can contain null, nulls are considered equal to one another but greater than all other nonnull values.

You must include a SEQUENCE BY clause if you include a sequence function in the select list of the SELECT statement. Otherwise, Trafodion SQL returns an error. Further, you cannot include a SEQUENCE BY clause if no sequence function exists in the select list. See "Sequence Functions" (page 282).

Considerations for SEQUENCE BY

- Sequence functions behave differently from set (or aggregate) functions and mathematical (or scalar) functions.
- If you include both SEQUENCE BY and GROUP BY clauses in the same SELECT statement, the values of the sequence functions must be evaluated first and then become input for aggregate functions in the statement.
  - For a SELECT statement that contains both SEQUENCE BY and GROUP BY clauses, you can nest the sequence function in the aggregate function:

    ```
    SELECT ordernum,
        MAX(MOVINGSUM(qty_ordered, 3)) AS maxmovsum_qty,
        AVG(unit_price) AS avg_price
    FROM odetail
    SEQUENCE BY partnum
    GROUP BY ordernum;
    ```

  - To use a sequence function as a grouping column, you must use a derived table for the SEQUENCE BY query and use the derived column in the GROUP BY clause:

    ```
    SELECT ordernum, movsum_qty, AVG(unit_price)
    FROM
      (SELECT ordernum, MOVINGSUM(qty_ordered, 3), unit_price
       FROM odetail
       SEQUENCE BY partnum)
         AS tab2 (ordernum, movsum_qty, unit_price)
    GROUP BY ordernum, movsum_qty;
    ```

  - To use an aggregate function as the argument to a sequence function, you must also use a derived table:

    ```
    SELECT MOVINGSUM(avg_price,2)
    FROM
      (SELECT ordernum, AVG(unit_price)
       FROM odetail
    ```

```
    GROUP BY ordernum)
      AS tab2 (ordernum, avg_price)
  SEQUENCE BY ordernum;
```

- Like aggregate functions, sequence functions generate an intermediate result. If the query has a WHERE clause, its search condition is applied during the generation of the intermediate result. Therefore, you cannot use sequence functions in the WHERE clause of a SELECT statement.
  - This query returns an error:

    ```
    SELECT ordernum, partnum, RUNNINGAVG(unit_price)
    FROM odetail
    WHERE ordernum > 800000 AND RUNNINGAVG(unit_price) > 350
    SEQUENCE BY qty_ordered;
    ```

  - Apply a search condition to the result of a sequence function, use a derived table for the SEQUENCE BY query, and use the derived column in the WHERE clause:

    ```
    SELECT ordernum, partnum, runavg_price
    FROM
      (SELECT ordernum, partnum, RUNNINGAVG(unit_price)
       FROM odetail
       SEQUENCE BY qty_ordered)
         AS tab2 (ordernum, partnum, runavg_price)
    WHERE ordernum > 800000 AND runavg_price > 350;
    ```

Examples of SEQUENCE BY

- Sequentially number each row for the entire result and also number the rows for each part number:

```
SELECT RUNNINGCOUNT(*) AS RCOUNT, MOVINGCOUNT(*,
  ROWS SINCE (d.partnum<>THIS(d.partnum)))
    AS MCOUNT,
  d.partnum
FROM orders o, odetail d
WHERE o.ordernum=d.ordernum
SEQUENCE BY d.partnum, o.order_date, o.ordernum
ORDER BY d.partnum, o.order_date, o.ordernum;

RCOUNT                MCOUNT                Part/Num
--------------------  --------------------  --------
                   1                     1       212
                   2                     2       212
                   3                     1       244
                   4                     2       244
                   5                     3       244
                 ...                   ...       ...
                  67                     1      7301
                  68                     2      7301
                  69                     3      7301
                  70                     4      7301

--- 70 row(s) selected.
```

- Show the orders for each date, the amount for each order item and the moving total for each order, and the running total of all the orders. The query sequences orders by date, order number, and part number. (The CAST function is used for readability only.)

```
SELECT o.ordernum,
  CAST (MOVINGCOUNT(*,ROWS SINCE(THIS(o.ordernum) <>
     o.ordernum)) AS INT) AS MCOUNT,
  d.partnum,  o.order_date,
  (d.unit_price * d.qty_ordered) AS AMOUNT,
  MOVINGSUM (d.unit_price * d.qty_ordered,
```

```
    ROWS SINCE(THIS(o.ordernum)<>o.ordernum)) AS ORDER_TOTAL,
  RUNNINGSUM (d.unit_price * d.qty_ordered) AS TOTAL_SALES
FROM orders o, odetail d
WHERE o.ordernum=d.ordernum
SEQUENCE BY o.order_date, o.ordernum, d.partnum
ORDER BY o.order_date, o.ordernum, d.partnum;

Order/Num   MCOUNT       Part/Num  Order/Date
AMOUNT      ORDER_TOTAL     TOTAL_SALES
----------  -----------  --------  ----------
----------  --------------  --------------

    100250            1       244  2008-01-23
  14000.00        14000.00          14000.00
    100250            2      5103  2008-01-23
   4000.00        18000.00          18000.00
    100250            3      6500  2008-01-23
    950.00        18950.00          18950.00

    200300            1       244  2008-02-06
  28000.00        28000.00          46950.00
    200300            2      2001  2008-02-06
  10000.00        38000.00          56950.00
    200300            3      2002  2008-02-06
  14000.00        52000.00          70950.00
       ...          ...         ...         ...
    800660           18      7102  2008-10-09
   1650.00       187360.00        1113295.00
    800660           19      7301  2008-10-09
   5100.00       192460.00        1118395.00

--- 69 row(s) selected.
```

For example, for order number 200300, the ORDER_TOTAL is a moving sum within the order date 2008-02-06, and the TOTAL_SALES is a running sum for all orders. The current window for the moving sum is defined as `ROWS SINCE (THIS(o.ordernum)<>o.ordernum)`, which restricts the ORDER_TOTAL to the current order number.

- Show the amount of time between orders by calculating the interval between two dates:

```
SELECT RUNNINGCOUNT(*),o.order_date,DIFF1(o.order_date)
FROM orders o
SEQUENCE BY o.order_date, o.ordernum
ORDER BY o.order_date, o.ordernum ;

(EXPR)                Order/Date  (EXPR)
--------------------  ----------  -------------
                   1  2008-01-23             ?
                   2  2008-02-06            14
                   3  2008-02-17            11
                   4  2008-03-03            14
                   5  2008-03-19            16
                   6  2008-03-19             0
                   7  2008-03-27             8
                   8  2008-04-10            14
                   9  2008-04-20            10
                  10  2008-05-12            22
                  11  2008-06-01            20
                  12  2008-07-21            50
                  13  2008-10-09            80

--- 13 row(s) selected.
```

TRANSPOSE Clause

- "Considerations for TRANSPOSE"
- "Examples of TRANSPOSE"

The TRANSPOSE clause of the SELECT statement generates for each row of the SELECT source table a row for each item in the transpose item list. The result table of the TRANSPOSE clause has all the columns of the source table plus, for each transpose item list, a value column or columns and an optional key column.

TRANSPOSE is a Trafodion SQL extension.

```
TRANSPOSE transpose-set [transpose-set]...
   [KEY BY key-colname]

transpose-set is:
   transpose-item-list AS transpose-col-list

transpose-item-list is:
     expression-list
   | (expression-list) [,(expression-list)]...

expression-list is:
   expression [,expression]...

transpose-col-list is:
     colname   | (colname-list)

colname-list is:
   colname [,colname]...
```

transpose-item-list AS *transpose-col-list*

specifies a *transpose-set*, which correlates a *transpose-item-list* with a *transpose-col-list*. The *transpose-item-list* can be a list of expressions or a list of expression lists enclosed in parentheses. The *transpose-col-list* can be a single column name or a list of column names enclosed in parentheses.

For example, in the *transpose-set* TRANSPOSE (A,X),(B,Y),(C,Z) AS (V1,V2), the items in the *transpose-item-list* are (A,X),(B,Y), and (C,Z), and the *transpose-col-list* is (V1,V2). The number of expressions in each item must be the same as the number of value columns in the column list.

In the example TRANSPOSE A,B,C AS V, the items are A,B, and C, and the value column is V. This form can be thought of as a shorter way of writing TRANSPOSE (A),(B),(C) AS (V).

transpose-item-list

specifies a list of items. An item is a value expression or a list of value expressions enclosed in parentheses.

expression-list

specifies a list of SQL value expressions, separated by commas. The expressions must have compatible data types.

For example, in the transpose set TRANSPOSE A,B,C AS V, the expressions A,B, and C have compatible data types.

(*expression-list*) [,(*expression-list*)]...

specifies a list of expressions enclosed in parentheses, followed by another list of expressions enclosed in parentheses, and so on. The number of expressions within parentheses must be equal for each list. The expressions in the same ordinal position within the parentheses must have compatible data types.

For example, in the transpose set TRANSPOSE (A,X),(B,Y),(C,Z) AS (V1,V2), the expressions A,B, and C have compatible data types, and the expressions X,Y, and Z have compatible data types.

transpose-col-list

specifies the columns that consist of the evaluation of expressions in the item list as the expressions are applied to rows of the source table.

colname

is an SQL identifier that specifies a column name. It identifies the column consisting of the values in *expression-list*.

For example, in the transpose set TRANSPOSE A,B,C AS V, the column V corresponds to the values of the expressions A,B, and C.

(*colname-list*)

specifies a list of column names enclosed in parentheses. Each column consists of the values of the expressions in the same ordinal position within the parentheses in the transpose item list.

For example, in the transpose set TRANSPOSE (A,X),(B,Y),(C,Z) AS (V1,V2), the column V1 corresponds to the expressions A,B, and C, and the column V2 corresponds to the expressions X,Y, and Z.

KEY BY *key-colname*

optionally specifies which expression (the value in the transpose column list corresponds to) by its position in the item list. *key-colname* is an SQL identifier. The data type of the key column is exact numeric, and the value is NOT NULL.

Considerations for TRANSPOSE

Multiple TRANSPOSE Clauses and Sets

- Multiple TRANSPOSE clauses can be used in the same query. For example:

```
SELECT KEYCOL1, VALCOL1, KEYCOL2, VALCOL2 FROM MYTABLE
TRANSPOSE A, B, C AS VALCOL1
  KEY BY KEYCOL1
TRANSPOSE D, E, F AS VALCOL2
  KEY BY KEYCOL2
```

- A TRANSPOSE clause can contain multiple transpose sets. For example:

```
SELECT KEYCOL, VALCOL1, VALCOL2 FROM MYTABLE
TRANSPOSE A, B, C AS VALCOL1
          D, E, F AS VALCOL2
  KEY BY KEYCOL
```

Degree and Column Order of the TRANSPOSE Result

The degree of the TRANSPOSE result is the degree of the source table (the result table derived from the table reference or references in the FROM clause and a WHERE clause if specified), plus one if the key column is specified, plus the cardinalities of all the transpose column lists.

The columns of the TRANSPOSE result are ordered beginning with the columns of the source table, followed by the key column if specified, and then followed by the list of column names in the order in which they are specified.

Data Type of the TRANSPOSE Result

The data type of each of the value columns is the union compatible data type of the corresponding expressions in the *transpose-item-list*. You cannot have expressions with data types that are not compatible in a *transpose-item-list*.

For example, in TRANSPOSE (A,X),(B,Y),(C,Z) AS (V1,V2), the data type of V1 is the union compatible type for A, B, and C, and the data type of V2 is the union compatible type for X, Y, and Z.

See "Comparable and Compatible Data Types" (page 201).

Cardinality of the TRANSPOSE Result

The items in each *transpose-item-list* are enumerated from 1 to N, where N is the total number of items in all the item lists in the transpose sets.

In this example with a single transpose set, the value of N is 3:

```
TRANSPOSE (A,X),(B,Y),(C,Z) AS (V1,V2)
```

In this example with two transpose sets, the value of N is 5:

```
TRANSPOSE (A,X),(B,Y),(C,Z) AS (V1,V2)
          L,M AS V3
```

The values 1 to N are the key values *k*i. The items in each *transpose-item-list* are the expression values *v*i.

The cardinality of the result of the TRANSPOSE clause is the cardinality of the source table times N, the total number of items in all the transpose item lists.

For each row of the source table and for each value in the key values *k*i, the TRANSPOSE result contains a row with all the attributes of the source table, the key value *k*i in the key column, the expression values vi in the value columns of the corresponding transpose set, and NULL in the value columns of other transpose sets.

For example, consider this TRANSPOSE clause:

```
TRANSPOSE (A,X),(B,Y),(C,Z) AS (V1,V2)
          L,M AS V3
  KEY BY K
```

The value of N is 5. One row of the SELECT source table produces this TRANSPOSE result:

columns-of-source	**K**	**V1**	**V2**	**V3**
source-row	1	*value-of-A*	*value-of-X*	NULL
source-row	2	*value-of-B*	*value-of-Y*	NULL
source-row	3	*value-of-C*	*value-of-Z*	NULL
source-row	4	NULL	NULL	*value-of-L*
source-row	5	NULL	NULL	*value-of-M*

Examples of TRANSPOSE

Suppose that MYTABLE has been created as:

```
CREATE TABLE mining.mytable
 ( A INTEGER, B INTEGER, C INTEGER, D CHAR(2),
   E CHAR(2), F CHAR(2) );
```

The table MYTABLE has columns A, B, C, D, E, and F with related data. The columns A, B, and C are type INTEGER, and columns D, E, and F are type CHAR.

A	B	C	D	E	F
1	10	100	d1	e1	f1
2	20	200	d2	e2	f2

- Suppose that MYTABLE has only the first three columns: A, B, and C. The result of the TRANSPOSE clause has three times as many rows (because three items exist in the transpose item list) as rows exist in MYTABLE:

```
SELECT * FROM mytable
TRANSPOSE A, B, C AS VALCOL
  KEY BY KEYCOL;
```

The result table of the TRANSPOSE query is:

A	B	C	D	E	F	KEYCOL	VALCOL
1	10	100	d1	e1	f1	1	1
1	10	100	d1	e1	f1	2	10
1	10	100	d1	e1	f1	3	100
2	20	200	d2	e2	f2	1	2
2	20	200	d2	e2	f2	2	20
2	20	200	d2	e2	f2	3	200

- This query shows that the items in the transpose item list can be any valid scalar expressions:

```
SELECT KEYCOL, VALCOL, A, B, C FROM mytable
TRANSPOSE A + B, C + 3, 6 AS VALCOL
  KEY BY KEYCOL;
```

The result table of the TRANSPOSE query is:

KEYCOL	**VALCOL**	**A**	**B**	**C**
1	11	1	10	100
2	103	1	10	100
3	6	1	10	100
1	22	2	20	200
2	203	2	20	200
3	6	2	20	200

- This query shows how the TRANSPOSE clause can be used with a GROUP BY clause. This query is typical of queries used to obtain cross-table information, where A, B, and C are the independent variables, and D is the dependent variable.

```
SELECT KEYCOL, VALCOL, D, COUNT(*) FROM mytable
TRANSPOSE A, B, C AS VALCOL
  KEY BY KEYCOL
GROUP BY KEYCOL, VALCOL, D;
```

The result table of the TRANSPOSE query is:

KEYCOL	VALCOL	D	COUNT(*)
1	1	d1	1
2	10	d1	1
3	100	d1	1
1	2	d2	1
2	20	d2	1
3	200	d2	1

- This query shows how to use COUNT applied to VALCOL. The result table of the TRANSPOSE query shows the number of distinct values in VALCOL.

```
SELECT COUNT(DISTINCT VALCOL) FROM mytable
TRANSPOSE A, B, C AS VALCOL
  KEY BY KEYCOL
GROUP BY KEYCOL;

(EXPR)
--------------------
                   2
                   2
                   2

--- 3 row(s) selected.
```

- This query shows how multiple TRANSPOSE clauses can be used in the same query. The result table from this query has nine times as many rows as rows exist in MYTABLE:

```
SELECT KEYCOL1, VALCOL1, KEYCOL2, VALCOL2 FROM mytable
TRANSPOSE A, B, C AS VALCOL1
  KEY BY KEYCOL1
TRANSPOSE D, E, F AS VALCOL2
  KEY BY KEYCOL2;
```

The result table of the TRANSPOSE query is:

KEYCOL1	VALCOL1	KEYCOL2	VALCOL2
1	1	1	d1
1	1	2	e1
1	1	3	f1
2	10	1	d1
2	10	2	e1
2	10	3	f1
3	100	1	d1
3	100	2	e1
3	100	3	f1
1	2	1	d2
1	2	2	e2
1	2	3	f2
2	20	1	d2
2	20	2	e2

KEYCOL1	VALCOL1	KEYCOL2	VALCOL2
2	20	3	f2
3	200	1	d2
3	200	2	e2
3	200	3	f2

- This query shows how a TRANSPOSE clause can contain multiple transpose sets—that is, multiple *transpose-item-list* AS *transpose-col-list*. The expressions A, B, and C are of type integer, and expressions D, E, and F are of type character.

```
SELECT KEYCOL, VALCOL1, VALCOL2 FROM mytable
TRANSPOSE A, B, C AS VALCOL1
          D, E, F AS VALCOL2
  KEY BY KEYCOL;
```

The result table of the TRANSPOSE query is:

KEYCOL	VALCOL1	VALCOL2
1	1	?
2	10	?
3	100	?
4	?	d1
5	?	e1
6	?	f1
1	2	?
2	20	?
3	200	?
4	?	d2
5	?	e2
6	?	f2

A question mark (?) in a value column indicates no value for the given KEYCOL.

- This query shows how the preceding query can include a GROUP BY clause:

```
SELECT KEYCOL, VALCOL1, VALCOL2, COUNT(*) FROM mytable
TRANSPOSE A, B, C AS VALCOL1
          D, E, F AS VALCOL2
  KEY BY KEYCOL
GROUP BY KEYCOL, VALCOL1, VALCOL2;
```

The result table of the TRANSPOSE query is:

KEYCOL	VALCOL1	VALCOL2	(EXPR)
1	1	?	1
2	10	?	1
3	100	?	1
1	2	?	1
2	20	?	1
3	200	?	1
4	?	d2	1

KEYCOL	VALCOL1	VALCOL2	(EXPR)
5	?	e2	1
6	?	f2	1
4	?	d1	1
5	?	e1	1
6	?	f1	1

- This query shows how an item in the transpose item list can contain a list of expressions and that the KEY BY clause is optional:

```
SELECT * FROM mytable
TRANSPOSE (1, A, 'abc'), (2, B, 'xyz')
  AS (VALCOL1, VALCOL2, VALCOL3);
```

The result table of the TRANSPOSE query is:

A	B	C	D	E	F	VALCOL1	VALCOL2	VALCOL3
1	10	100	d1	e1	f1	1	1	abc
1	10	100	d1	e1	f1	2	10	xyz
2	20	200	d2	e2	f2	1	2	abc
2	20	200	d2	e2	f2	2	20	xyz

6 SQL Functions and Expressions

This section describes the syntax and semantics of specific functions and expressions that you can use in Trafodion SQL statements. The functions and expressions are categorized according to their functionality.

Categories

Use these types of functions within an SQL value expression:

- "Aggregate (Set) Functions"
- "Character String Functions"
- "Datetime Functions"
- "Mathematical Functions"
- "Sequence Functions"
- "Other Functions and Expressions"

For more information on SQL value expressions, see "Expressions" (page 211).

Standard Normalization

For datetime functions, the definition of standard normalization is: If the ending day of the resulting date is invalid, the day will be rounded DOWN to the last day of the result month.

Aggregate (Set) Functions

An aggregate (or set) function operates on a group or groups of rows retrieved by the SELECT statement or the subquery in which the aggregate function appears.

"AVG Function" (page 293)	Computes the average of a group of numbers derived from the evaluation of the expression argument of the function.
"COUNT Function" (page 313)	Counts the number of rows that result from a query (by using *) or the number of rows that contain a distinct value in the one-column table derived from the expression argument of the function (optionally distinct values).
"MAX/MAXIMUM Function" (page 360)	Determines a maximum value from the group of values derived from the evaluation of the expression argument.
"MIN Function" (page 361)	Determines a minimum value from the group of values derived from the evaluation of the expression argument.
"STDDEV Function" (page 410)	Computes the statistical standard deviation of a group of numbers derived from the evaluation of the expression argument of the function. The numbers can be weighted.
"SUM Function" (page 414)	Computes the sum of a group of numbers derived from the evaluation of the expression argument of the function.
"VARIANCE Function" (page 426)	Computes the statistical variance of a group of numbers derived from the evaluation of the expression argument of the function. The numbers can be weighted.

Columns and expressions can be arguments of an aggregate function. The expressions cannot contain aggregate functions or subqueries.

An aggregate function can accept an argument specified as DISTINCT, which eliminates duplicate values before the aggregate function is applied. See "DISTINCT Aggregate Functions" (page 147).

If you include a GROUP BY clause in the SELECT statement, the columns you refer to in the select list must be either grouping columns or arguments of an aggregate function. If you do not include

a GROUP BY clause but you specify an aggregate function in the select list, all rows of the SELECT result table form the one and only group.

See the individual entry for the function.

Character String Functions

These functions manipulate character strings and use a character value expression as an argument or return a result of a character data type. Character string functions treat each single-byte or multibyte character in an input string as one character, regardless of the byte length of the character.

"ASCII Function" (page 288)	Returns the ASCII code value of the first character of a character value expression.
"CHAR Function" (page 302)	Returns the specified code value in a character set.
"CHAR_LENGTH Function" (page 303)	Returns the number of characters in a string. You can also use CHARACTER_LENGTH.
"CODE_VALUE Function" (page 305)	Returns an unsigned integer that is the code point of the first character in a character value expression that can be associated with one of the supported character sets.
"CONCAT Function" (page 306)	Returns the concatenation of two character value expressions as a string value. You can also use the concatenation operator (||).
"INSERT Function" (page 348)	Returns a character string where a specified number of characters within the character string have been deleted and then a second character string has been inserted at a specified start position.
"LCASE Function" (page 352)	Downshifts alphanumeric characters. You can also use LOWER.
"LEFT Function" (page 353)	Returns the leftmost specified number of characters from a character expression.
"LOCATE Function" (page 354)	Returns the position of a specified substring within a character string. You can also use POSITION.
"LOWER Function" (page 357)	Downshifts alphanumeric characters. You can also use LCASE.
"LPAD Function" (page 358)	Replaces the leftmost specified number of characters in a character expression with a padding character.
"LTRIM Function" (page 359)	Removes leading spaces from a character string.
"OCTET_LENGTH Function" (page 378)	Returns the length of a character string in bytes.
"POSITION Function" (page 381)	Returns the position of a specified substring within a character string. You can also use LOCATE.
"REPEAT Function" (page 388)	Returns a character string composed of the evaluation of a character expression repeated a specified number of times.
"REPLACE Function" (page 389)	Returns a character string where all occurrences of a specified character string in the original string are replaced with another character string.
"RIGHT Function" (page 390)	Returns the rightmost specified number of characters from a character expression.
"RPAD Function" (page 395)	Replaces the rightmost specified number of characters in a character expression with a padding character.
"RTRIM Function" (page 396)	Removes trailing spaces from a character string.
"SPACE Function" (page 408)	Returns a character string consisting of a specified number of spaces.
"SUBSTRING/SUBSTR Function" (page 412)	Extracts a substring from a character string.

"TRANSLATE Function" (page 420)	Translates a character string from a source character set to a target character set.
"TRIM Function" (page 421)	Removes leading or trailing characters from a character string.
"UCASE Function" (page 422)	Upshifts alphanumeric characters. You can also use UPSHIFT or UPPER.
"UPPER Function" (page 423)	Upshifts alphanumeric characters. You can also use UPSHIFT or UCASE.
"UPSHIFT Function" (page 424)	Upshifts alphanumeric characters. You can also use UPPER or UCASE.

See the individual entry for the function.

Datetime Functions

These functions use either a datetime value expression as an argument or return a result of datetime data type:

"ADD_MONTHS Function" (page 287)	Adds the integer number of months specified by *intr_expr* to *datetime_expr* and normalizes the result.
"CONVERTTIMESTAMP Function" (page 310)	Converts a Julian timestamp to a TIMESTAMP value.
"CURRENT Function" (page 315)	Returns the current timestamp. You can also use the "CURRENT_TIMESTAMP Function".
"CURRENT_DATE Function" (page 316)	Returns the current date.
"CURRENT_TIME Function" (page 317)	Returns the current time.
"CURRENT_TIMESTAMP Function" (page 318)	Returns the current timestamp. You can also use the "CURRENT Function".
"DATE_ADD Function" (page 320)	Adds the interval specified by *interval_expression* to *datetime_expr*.
"DATE_PART Function (of an Interval)" (page 325)	Extracts the datetime field specified by *text* from the interval value specified by `interval` and returns the result as an exact numeric value.
"DATE_PART Function (of a Timestamp)" (page 326)	Extracts the datetime field specified by *text* from the datetime value specified by `timestamp` and returns the result as an exact numeric value.
"DATE_SUB Function" (page 321)	Subtracts the specified *interval_expression* from *datetime_expr*.
"DATE_TRUNC Function" (page 327)	Returns the date with the time portion of the day truncated.
"DATEADD Function" (page 322)	Adds the interval specified by *datepart* and *num_expr* to *datetime_expr*.
"DATEDIFF Function" (page 323)	Returns the integer value for the number of *datepart* units of time between *startdate* and *enddate*.
"DATEFORMAT Function" (page 324)	Formats a datetime value for display purposes.
"DAY Function" (page 328)	Returns an integer value in the range 1 through 31 that represents the corresponding day of the month. You can also use DAYOFMONTH.
"DAYNAME Function" (page 329)	Returns the name of the day of the week from a date or timestamp expression.
"DAYOFMONTH Function" (page 330)	Returns an integer value in the range 1 through 31 that represents the corresponding day of the month. You can also use DAY.
"DAYOFWEEK Function" (page 331)	Returns an integer value in the range 1 through 7 that represents the corresponding day of the week.

"DAYOFYEAR Function" (page 332)	Returns an integer value in the range 1 through 366 that represents the corresponding day of the year.
"EXTRACT Function" (page 345)	Returns a specified datetime field from a datetime value expression or an interval value expression.
"HOUR Function" (page 347)	Returns an integer value in the range 0 through 23 that represents the corresponding hour of the day.
"JULIANTIMESTAMP Function" (page 350)	Converts a datetime value to a Julian timestamp.
"MINUTE Function" (page 362)	Returns an integer value in the range 0 through 59 that represents the corresponding minute of the hour.
"MONTH Function" (page 364)	Returns an integer value in the range 1 through 12 that represents the corresponding month of the year.
"MONTHNAME Function" (page 365)	Returns a character literal that is the name of the month of the year (January, February, and so on).
"QUARTER Function" (page 383)	Returns an integer value in the range 1 through 4 that represents the corresponding quarter of the year.
"SECOND Function" (page 404)	Returns an integer value in the range 0 through 59 that represents the corresponding second of the minute.
"TIMESTAMPADD Function" (page 418)	Adds the interval of time specified by *interval-ind* and *num_expr* to *datetime_expr*.
"TIMESTAMPDIFF Function" (page 419)	Returns the integer value for the number of *interval-ind* units of time between *startdate* and *enddate*.
"WEEK Function" (page 428)	Returns an integer value in the range 1 through 54 that represents the corresponding week of the year.
"YEAR Function" (page 429)	Returns an integer value that represents the year.

See the individual entry for the function.

Mathematical Functions

Use these mathematical functions within an SQL numeric value expression:

"ABS Function" (page 285)	Returns the absolute value of a numeric value expression.
"ACOS Function" (page 286)	Returns the arccosine of a numeric value expression as an angle expressed in radians.
"ASIN Function" (page 289)	Returns the arcsine of a numeric value expression as an angle expressed in radians.
"ATAN Function" (page 290)	Returns the arctangent of a numeric value expression as an angle expressed in radians.
"ATAN2 Function" (page 291)	Returns the arctangent of the x and y coordinates, specified by two numeric value expressions, as an angle expressed in radians.
"CEILING Function" (page 301)	Returns the smallest integer greater than or equal to a numeric value expression.
"COS Function" (page 311)	Returns the cosine of a numeric value expression, where the expression is an angle expressed in radians.
"COSH Function" (page 312)	Returns the hyperbolic cosine of a numeric value expression, where the expression is an angle expressed in radians.
"DEGREES Function" (page 336)	Converts a numeric value expression expressed in radians to the number of degrees.
"EXP Function" (page 341)	Returns the exponential value (to the base e) of a numeric value expression.
"FLOOR Function" (page 346)	Returns the largest integer less than or equal to a numeric value expression.

"LOG Function" (page 355)	Returns the natural logarithm of a numeric value expression.
"LOG10 Function" (page 356)	Returns the base 10 logarithm of a numeric value expression.
"MOD Function" (page 363)	Returns the remainder (modulus) of an integer value expression divided by an integer value expression.
"NULLIFZERO Function" (page 376)	Returns the value of the operand unless it is zero, in which case it returns NULL.
"PI Function" (page 380)	Returns the constant value of pi as a floating-point value.
"POWER Function" (page 382)	Returns the value of a numeric value expression raised to the power of an integer value expression. You can also use the exponential operator **.
"RADIANS Function" (page 384)	Converts a numeric value expression expressed in degrees to the number of radians.
"ROUND Function" (page 391)	Returns the value of *numeric_expr* round to *num* places to the right of the decimal point.
"SIGN Function" (page 405)	Returns an indicator of the sign of a numeric value expression. If value is less than zero, returns -1 as the indicator. If value is zero, returns 0. If value is greater than zero, returns 1.
"SIN Function" (page 406)	Returns the sine of a numeric value expression, where the expression is an angle expressed in radians.
"SINH Function" (page 407)	Returns the hyperbolic sine of a numeric value expression, where the expression is an angle expressed in radians.
"SQRT Function" (page 409)	Returns the square root of a numeric value expression.
"TAN Function" (page 415)	Returns the tangent of a numeric value expression, where the expression is an angle expressed in radians.
"TANH Function" (page 416)	Returns the hyperbolic tangent of a numeric value expression, where the expression is an angle expressed in radians.
"ZEROIFNULL Function" (page 430)	Returns the value of the operand unless it is NULL, in which case it returns zero.

See the individual entry for the function.

Sequence Functions

Sequence functions operate on ordered rows of the intermediate result table of a SELECT statement that includes a SEQUENCE BY clause. Sequence functions are categorized generally as difference, moving, offset, or running.

Some sequence functions, such as ROWS SINCE, require sequentially examining every row in the history buffer until the result is computed. Examining a large history buffer in this manner for a condition that has not been true for many rows could be an expensive operation. In addition, such operations may not be parallelized because the entire sorted result set must be available to compute the result of the sequence function.

Difference sequence functions:	
"DIFF1 Function" (page 337)	Calculates differences between values of a column expression in the current row and previous rows.
"DIFF2 Function" (page 339)	Calculates differences between values of the result of DIFF1 of the current row and DIFF1 of previous rows.
Moving sequence functions:	
"MOVINGAVG Function" (page 366)	Returns the average of nonnull values of a column expression in the current window.

"MOVINGCOUNT Function" (page 367)	Returns the number of nonnull values of a column expression in the current window.
"MOVINGMAX Function" (page 368)	Returns the maximum of nonnull values of a column expression in the current window.
"MOVINGMIN Function" (page 369)	Returns the minimum of nonnull values of a column expression in the current window.
"MOVINGSTDDEV Function" (page 370)	Returns the standard deviation of nonnull values of a column expression in the current window.
"MOVINGSUM Function" (page 372)	Returns the sum of nonnull values of a column expression in the current window.
"MOVINGVARIANCE Function" (page 373)	Returns the variance of nonnull values of a column expression in the current window.
Offset sequence function:	
"OFFSET Function" (page 379)	Retrieves columns from previous rows.
Running sequence functions:	
"RANK/RUNNINGRANK Function" (page 385)	Returns the rank of the given value of an intermediate result table ordered by a SEQUENCE BY clause in a SELECT statement
"RUNNINGAVG Function" (page 397)	Returns the average of nonnull values of a column expression up to and including the current row.
"RUNNINGCOUNT Function" (page 398)	Returns the number of rows up to and including the current row.
"RUNNINGMAX Function" (page 399)	Returns the maximum of values of a column expression up to and including the current row.
"RUNNINGMIN Function" (page 400)	Returns the minimum of values of a column expression up to and including the current row.
"RUNNINGRANK Function" (page 400)	Returns the rank of the given value of an intermediate result table ordered by a SEQUENCE BY clause in a SELECT statement.
"RUNNINGSTDDEV Function" (page 401)	Returns the standard deviation of nonnull values of a column expression up to and including the current row.
"RUNNINGSUM Function" (page 402)	Returns the sum of nonnull values of a column expression up to and including the current row.
"RUNNINGVARIANCE Function" (page 403)	Returns the variance of nonnull values of a column expression up to and including the current row.
Other sequence functions:	
"LASTNOTNULL Function" (page 351)	Returns the last nonnull value for the specified column expression. If only null values have been returned, returns null.
"ROWS SINCE Function" (page 392)	Returns the number of rows counted since the specified condition was last true.
"ROWS SINCE CHANGED Function" (page 394)	Returns the number of rows counted since the specified set of values last changed.
"THIS Function" (page 417)	Used in ROWS SINCE to distinguish between the value of the column in the current row and the value of the column in previous rows.

See "SEQUENCE BY Clause" (page 268) and the individual entry for each function.

Other Functions and Expressions

Use these other functions and expressions in an SQL value expression:

"AUTHNAME Function" (page 292)	Returns the authorization name associated with the specified authorization ID number.
"BITAND Function" (page 295)	Performs 'and' operation on corresponding bits of the two operands.
"CASE (Conditional) Expression" (page 296)	A conditional expression. The two forms of the CASE expression are simple and searched.
"CAST Expression" (page 299)	Converts a value from one data type to another data type that you specify.
"COALESCE Function" (page 304)	Returns the value of the first expression in the list that does not have a NULL value or if all the expressions have NULL values, the function returns a NULL value.
"CONVERTTOHEX Function" (page 308)	Converts the specified value expression to hexadecimal for display purposes.
"CURRENT_USER Function" (page 319)	Returns the database username of the current user who invoked the function.
"DECODE Function" (page 333)	Compares *expr* to each *test_expr* value one by one in the order provided.
"EXPLAIN Function" (page 342)	Generates a result table describing an access plan for a SELECT, INSERT, DELETE, or UPDATE statement.
"ISNULL Function" (page 349)	Returns the first argument if it is not null, otherwise it returns the second argument.
"NULLIF Function" (page 375)	Returns the value of the first operand if the two operands are not equal, otherwise it returns NULL.
"NVL Function" (page 377)	Returns the value of the first operand unless it is NULL, in which case it returns the value of the second operand.
"USER Function" (page 425)	Returns either the database username of the current user who invoked the function or the database username associated with the specified user ID number.

See the individual entry for the function.

ABS Function

The ABS function returns the absolute value of a numeric value expression.

ABS is a Trafodion SQL extension.

```
ABS (numeric-expression)
```

numeric-expression

is an SQL numeric value expression that specifies the value for the argument of the ABS function. The result is returned as an unsigned numeric value if the precision of the argument is less than 10 or as a LARGEINT if the precision of the argument is greater than or equal to 10. See "Numeric Value Expressions" (page 218).

Example of ABS

This function returns the value 8:

```
ABS (-20 + 12)
```

ACOS Function

The ACOS function returns the arccosine of a numeric value expression as an angle expressed in radians.

ACOS is a Trafodion SQL extension.

```
ACOS (numeric-expression)
```

numeric-expression

is an SQL numeric value expression that specifies the value for the argument of the ACOS function. The range for the value of the argument is from -1 to +1. See "Numeric Value Expressions" (page 218).

Examples of ACOS

- The ACOS function returns the value 3.49044274380724416E-001 or approximately 0.3491 in radians (which is 20 degrees).

  ```
  ACOS (0.9397)
  ```

- This function returns the value 0.3491. The function ACOS is the inverse of the function COS.

  ```
  ACOS (COS (0.3491))
  ```

ADD_MONTHS Function

The ADD_MONTHS function adds the integer number of months specified by *int_expr* to *datetime_expr* and normalizes the result.

ADD_MONTHS is a Trafodion SQL extension.

```
ADD_MONTHS (datetime_expr, int_expr [, int2  ])
```

datetime_expr

is an expression that evaluates to a datetime value of type DATE or TIMESTAMP. The return value is the same type as the *datetime_expr*. See "Datetime Value Expressions" (page 212).

int_expr

is an SQL numeric value expression of data type SMALLINT or INTEGER that specifies the number of months. See "Numeric Value Expressions" (page 218).

int2

is an unsigned integer constant. If *int2* is omitted or is the literal 0, the normalization is the standard normalization. If *int2* is the literal 1, the normalization includes the standard normalization and if the starting day (the day part of *datetime_expr*) is the last day of the starting month, then the ending day (the day part of the result value) is set to the last valid day of the result month. See "Standard Normalization" (page 278). See "Numeric Value Expressions" (page 218).

Examples of ADD_MONTHS

- This function returns the value DATE '2007-03-31':

  ```
  ADD_MONTHS(DATE '2007-02-28', 1, 1)
  ```

- This function returns the value DATE '2007-03-28':

  ```
  ADD_MONTHS(DATE '2007-02-28', 1, 0)
  ```

- This function returns the value DATE '2008-03-28':

  ```
  ADD_MONTHS(DATE '2008-02-28', 1, 1)
  ```

- This function returns the timestamp '2009-02-28 00:00:00':

  ```
  ADD_MONTHS(timestamp'2008-02-29 00:00:00',12,1)
  ```

ASCII Function

The ASCII function returns the integer that is the ASCII code of the first character in a character string expression associated with either the ISO8891 character set or the UTF8 character set.

ASCII is a Trafodion SQL extension.

```
ASCII (character-expression)
```

character-expression

is an SQL character value expression that specifies a string of characters. See "Character Value Expressions" (page 211).

Considerations for ASCII

For a string expression in the UTF8 character set, if the value of the first byte in the string is greater than 127, Trafodion SQL returns this error message:

```
ERROR[8428] The argument to function ASCII is not valid.
```

Example of ASCII

Select the column JOBDESC and return the ASCII code of the first character of the job description:

```
SELECT jobdesc, ASCII (jobdesc)
FROM persnl.job;

JOBDESC             (EXPR)
-----------------   --------
MANAGER             77
PRODUCTION SUPV     80
ASSEMBLER           65
SALESREP            83
...                 ...

--- 10 row(s) selected.
```

ASIN Function

The ASIN function returns the arcsine of a numeric value expression as an angle expressed in radians.

ASIN is a Trafodion SQL extension.

```
ASIN (numeric-expression)
```

numeric-expression

is an SQL numeric value expression that specifies the value for the argument of the ASIN function. The range for the value of the argument is from -1 to +1. See "Numeric Value Expressions" (page 218).

Examples of ASIN

- This function returns the value 3.49044414403046400E-001 or approximately 0.3491 in radians (which is 20 degrees):

  ```
  ASIN (0.3420)
  ```

- This function returns the value 0.3491. The function ASIN is the inverse of the function SIN.

  ```
  ASIN (SIN (0.3491))
  ```

ATAN Function

The ATAN function returns the arctangent of a numeric value expression as an angle expressed in radians.

ATAN is a Trafodion SQL extension.

```
ATAN (numeric-expression)
```

numeric-expression

is an SQL numeric value expression that specifies the value for the argument of the ATAN function. See "Numeric Value Expressions" (page 218).

Examples of ATAN

- This function returns the value 8.72766423249958272E-001 or approximately 0.8727 in radians (which is 50 degrees):

  ```
  ATAN (1.192)
  ```

- This function returns the value 0.8727. The function ATAN is the inverse of the function TAN.

  ```
  ATAN (TAN (0.8727))
  ```

ATAN2 Function

The ATAN2 function returns the arctangent of the x and y coordinates, specified by two numeric value expressions, as an angle expressed in radians.

ATAN2 is a Trafodion SQL extension.

```
ATAN2 (numeric-expression-x,numeric-expression-y)
```

numeric-expression-x, *numeric-expression-y*

are SQL numeric value expressions that specify the value for the x and y coordinate arguments of the ATAN2 function. See "Numeric Value Expressions" (page 218).

Example of ATAN2

This function returns the value 2.66344329881899520E+000, or approximately 2.6634:

```
ATAN2 (1.192,-2.3)
```

AUTHNAME Function

The AUTHNAME function returns the name of the authorization ID that is associated with the specified authorization ID number.

```
AUTHNAME (auth-id)
```

auth-id

is the 32-bit number associated with an authorization ID. See "Authorization IDs" (page 193).

The AUTHNAME function is similar to the "USER Function" (page 425).

Considerations for AUTHNAME

- This function can be specified only in the top level of a SELECT statement.
- The value returned is string data type VARCHAR(128) and is in ISO8859-1 encoding.

Example of AUTHNAME

This example shows the authorization name associated with the authorization ID number, 33333:

```
>>SELECT AUTHNAME (33333) FROM (values(1)) x(a);
(EXPR)
------------------------
DB__ROOT
--- 1 row(s) selected.
```

AVG Function

AVG is an aggregate function that returns the average of a set of numbers.

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

ALL | DISTINCT

specifies whether duplicate values are included in the computation of the AVG of the *expression*. The default option is ALL, which causes duplicate values to be included. If you specify DISTINCT, duplicate values are eliminated before the AVG function is applied.

expression

specifies a numeric or interval value *expression* that determines the values to average. The *expression* cannot contain an aggregate function or a subquery. The DISTINCT clause specifies that the AVG function operates on distinct values from the one-column table derived from the evaluation of *expression*.

See "Numeric Value Expressions" (page 218) and "Interval Value Expressions" (page 215).

Considerations for AVG

Data Type of the Result

The data type of the result depends on the data type of the argument. If the argument is an exact numeric type, the result is LARGEINT. If the argument is an approximate numeric type, the result is DOUBLE PRECISION. If the argument is INTERVAL data type, the result is INTERVAL with the same precision as the argument.

The scale of the result is the same as the scale of the argument. If the argument has no scale, the result is truncated.

Operands of the Expression

The expression includes columns from the rows of the SELECT result table but cannot include an aggregate function. These expressions are valid:

```
AVG (SALARY)
AVG (SALARY * 1.1)
AVG (PARTCOST * QTY_ORDERED)
```

Nulls

All nulls are eliminated before the function is applied to the set of values. If the result table is empty, AVG returns NULL.

Examples of AVG

- Return the average value of the SALARY column:

```
SELECT AVG (salary)
FROM persnl.employee;

(EXPR)
----------------------
              49441.52

--- 1 row(s) selected.
```

- Return the average value of the set of unique SALARY values:

```
SELECT AVG(DISTINCT salary) AS Avg_Distinct_Salary
FROM persnl.employee;

AVG_DISTINCT_SALARY
----------------------
              53609.89
```

```
--- 1 row(s) selected.
```

- Return the average salary by department:

```
SELECT deptnum, AVG (salary) AS "AVERAGE SALARY"
FROM persnl.employee
WHERE deptnum < 3000
GROUP BY deptnum;

Dept/Num  "AVERAGE SALARY"
--------  --------------------
    1000               52000.17
    2000               50000.10
    1500               41250.00
    2500               37000.00

--- 4 row(s) selected.
```

BITAND Function

The BITAND function performs an AND operation on corresponding bits of the two operands. If both bits are 1, the result bit is 1. Otherwise the result bit is 0.

```
BITAND (expression, expression)
```

expression

The result data type is a binary number. Depending on the precision of the operands, the data type of the result can either be an INT (32-bit integer) or a LARGEINT (64-bit integer).

If the max precision of either operand is greater than 9, LARGEINT is chosen (numbers with precision greater than 9 are represented by LARGEINT). Otherwise, INT is chosen.

If both operands are unsigned, the result is unsigned. Otherwise, the result is signed.

Both operands are converted to the result data type before performing the bit operation.

Considerations for BITAND

BITAND can be used anywhere in an SQL query where an expression could be used. This includes SELECT lists, WHERE predicates, VALUES clauses, SET statement, and so on.

This function returns a numeric data type and can be used in arithmetic expressions.

Numeric operands can be positive or negative numbers. All numeric data types are allowed with the exceptions listed in the "Restrictions for BITAND" section.

Restrictions for BITAND

The following are BITAND restrictions:

- Must have 2 operands
- Operands must be binary or decimal exact numerics
- Operands must have scale of zero
- Operands cannot be floating point numbers
- Operands cannot be an extended precision numeric (the maximum precision of an extended numeric data type is 128)

Examples of BITAND

```
>>select bitand(1,3) from (values(1)) x(a);

(EXPR)
--------------
               1

--- 1 row(s) selected
>>select 1 & 3 from (values(1)) x(a);

(EXPR)
--------------
               1

--- 1 row(s) selected
>>select bitand(1,3) + 0 from (values(1)) x(a);

(EXPR)
--------------
               1

--- 1 row(s) selected
```

CASE (Conditional) Expression

- "Considerations for CASE"
- "Examples of CASE"

The CASE expression is a conditional expression with two forms: simple and searched.

In a simple CASE expression, Trafodion SQL compares a value to a sequence of values and sets the CASE expression to the value associated with the first match—if a match exists. If no match exists, Trafodion SQL returns the value specified in the ELSE clause (which can be null).

In a searched CASE expression, Trafodion SQL evaluates a sequence of conditions and sets the CASE expression to the value associated with the first condition that is true—if a true condition exists. If no true condition exists, Trafodion SQL returns the value specified in the ELSE clause (which can be null).

```
Simple CASE is:

CASE case-expression
  WHEN expression-1 THEN {result-expression-1 | NULL}
  WHEN expression-2 THEN {result-expression-2 | NULL}
  ...
  WHEN expression-n THEN {result-expression-n | NULL}
  [ELSE {result-expression | NULL}]
END

Searched CASE is:

CASE
  WHEN condition-1 THEN {result-expression-1 | NULL}
  WHEN condition-2 THEN {result-expression-2 | NULL}
  ...
  WHEN condition-n THEN {result-expression-n | NULL}
  [ELSE {result-expression | NULL}]
END
```

case-expression

specifies a value expression that is compared to the value expressions in each WHEN clause of a simple CASE. The data type of each *expression* in the WHEN clause must be comparable to the data type of *case-expression*.

expression-1 ... *expression-n*

specifies a value associated with each *result-expression*. If the value of an *expression* in a WHEN clause matches the value of *case-expression*, simple CASE returns the associated *result-expression* value. If no match exists, the CASE expression returns the value expression specified in the ELSE clause, or NULL if the ELSE value is not specified.

result-expression-1 ... *result-expression-n*

specifies the result value expression associated with each *expression* in a WHEN clause of a simple CASE, or with each *condition* in a WHEN clause of a searched CASE. All of the *result-expressions* must have comparable data types, and at least one of the *result-expressions* must return nonnull.

result-expression

follows the ELSE keyword and specifies the value returned if none of the expressions in the WHEN clause of a simple CASE are equal to the case expression, or if none of the conditions in the WHEN clause of a searched CASE are true. If the ELSE *result-expression* clause is not specified, CASE returns NULL. The data type of *result-expression* must be comparable to the other results.

condition-1 ... *condition-n*

specifies conditions to test for in a searched CASE. If a *condition* is true, the CASE expression returns the associated *result-expression* value. If no *condition* is true, the CASE expression returns the value expression specified in the ELSE clause, or NULL if the ELSE value is not specified.

Considerations for CASE

Data Type of the CASE Expression

The data type of the result of the CASE expression depends on the data types of the result expressions. If the results all have the same data type, the CASE expression adopts that data type. If the results have comparable but not identical data types, the CASE expression adopts the data type of the union of the result expressions. This result data type is determined in these ways.

Character Data Type

If any data type of the result expressions is variable-length character string, the result data type is variable-length character string with maximum length equal to the maximum length of the result expressions.

Otherwise, if none of the data types is variable-length character string, the result data type is fixed-length character string with length equal to the maximum of the lengths of the result expressions.

Numeric Data Type

If all of the data types of the result expressions are exact numeric, the result data type is exact numeric with precision and scale equal to the maximum of the precisions and scales of the result expressions.

For example, if *result-expression-1* and *result-expression-2* have data type NUMERIC(5) and *result-expression-3* has data type NUMERIC(8,5), the result data type is NUMERIC(10,5).

If any data type of the result expressions is approximate numeric, the result data type is approximate numeric with precision equal to the maximum of the precisions of the result expressions.

Datetime Data Type

If the data type of the result expressions is datetime, the result data type is the same datetime data type.

Interval Data Type

If the data type of the result expressions is interval, the result data type is the same interval data type (either year-month or day-time) with the start field being the most significant of the start fields of the result expressions and the end field being the least significant of the end fields of the result expressions.

Examples of CASE

- Use a simple CASE to decode JOBCODE and return NULL if JOBCODE does not match any of the listed values:

```
SELECT last_name, first_name,
  CASE jobcode
    WHEN 100 THEN 'MANAGER'
    WHEN 200 THEN 'PRODUCTION SUPV'
    WHEN 250 THEN 'ASSEMBLER'
    WHEN 300 THEN 'SALESREP'
    WHEN 400 THEN 'SYSTEM ANALYST'
    WHEN 420 THEN 'ENGINEER'
    WHEN 450 THEN 'PROGRAMMER'
```

```
    WHEN 500 THEN 'ACCOUNTANT'
    WHEN 600 THEN 'ADMINISTRATOR ANALYST'
    WHEN 900 THEN 'SECRETARY'
    ELSE NULL
  END
FROM persnl.employee;

LAST_NAME             FIRST_NAME       (EXPR)
--------------------  ---------------  -----------------

GREEN                 ROGER            MANAGER
HOWARD                JERRY            MANAGER
RAYMOND               JANE             MANAGER
...
CHOU                  JOHN             SECRETARY
CONRAD                MANFRED          PROGRAMMER
HERMAN                JIM              SALESREP
CLARK                 LARRY            ACCOUNTANT
HALL                  KATHRYN          SYSTEM ANALYST
...

--- 62 row(s) selected.
```

- Use a searched CASE to return LAST_NAME, FIRST_NAME and a value based on SALARY that depends on the value of DEPTNUM:

```
SELECT last_name, first_name, deptnum,
  CASE
    WHEN deptnum = 9000 THEN salary * 1.10
    WHEN deptnum = 1000 THEN salary * 1.12
    ELSE salary
  END
FROM persnl.employee;

LAST_NAME        FIRST_NAME    DEPTNUM  (EXPR)
---------------- ------------  -------  -------------------
GREEN            ROGER            9000          193050.0000
HOWARD           JERRY            1000          153440.1120
RAYMOND          JANE             3000          136000.0000
...
--- 62 row(s) selected.
```

CAST Expression

- "Considerations for CAST"
- "Valid Conversions for CAST "
- "Examples of CAST"

The CAST expression converts data to the data type you specify.

```
CAST ({expression | NULL} AS data-type)
```

expression | NULL

specifies the operand to convert to the data type *data-type*.

If the operand is an *expression*, then *data-type* depends on the data type of *expression* and follows the rules outlined in "Valid Conversions for CAST " (page 299).

If the operand is NULL, or if the value of the *expression* is null, the result of CAST is NULL, regardless of the data type you specify.

data-type

specifies a data type to associate with the operand of CAST. See "Data Types" (page 199).

When casting data to a CHAR or VARCHAR data type, the resulting data value is left justified. Otherwise, the resulting data value is right justified. Further, when you are casting to a CHAR or VARCHAR data type, you must specify the length of the target value.

Considerations for CAST

- Fractional portions are discarded when you use CAST of a numeric value to an INTERVAL type.
- Depending on how your file is set up, using CAST might cause poor query performance by preventing the optimizer from choosing the most efficient plan and requiring the executor to perform a complete table or index scan.

Valid Conversions for CAST

- An exact or approximate numeric value to any other numeric data type.
- An exact or approximate numeric value to any character string data type.
- An exact numeric value to either a single-field year-month or day-time interval such as INTERVAL DAY(2).
- A character string to any other data type, with one restriction:

  The contents of the character string to be converted must be consistent in meaning with the data type of the result. For example, if you are converting to DATE, the contents of the character string must be 10 characters consisting of the year, a hyphen, the month, another hyphen, and the day.
- A date value to a character string or to a TIMESTAMP (Trafodion SQL fills in the time part with 00:00:00.00).
- A time value to a character string or to a TIMESTAMP (Trafodion SQL fills in the date part with the current date).
- A timestamp value to a character string, a DATE, a TIME, or another TIMESTAMP with different fractional seconds precision.
- A year-month interval value to a character string, an exact numeric, or to another year-month INTERVAL with a different start field precision.
- A day-time interval value to a character string, an exact numeric, or to another day-time INTERVAL with a different start field precision.

Examples of CAST

- In this example, the fractional portion is discarded:

  ```
  CAST (123.956 as INTERVAL DAY(18))
  ```

- This example returns the difference of two timestamps in minutes:

  ```
  CAST((d.step_end - d.step_start) AS INTERVAL MINUTE)
  ```

- Suppose that your database includes a log file of user information. This example converts the current timestamp to a character string and concatenates the result to a character literal. Note the length must be specified.

  ```
  INSERT INTO stats.logfile
  (user_key, user_info)
  VALUES (001, 'User JBrook, executed at ' ||
          CAST (CURRENT_TIMESTAMP AS CHAR(26)));
  ```

CEILING Function

The CEILING function returns the smallest integer, represented as a FLOAT data type, greater than or equal to a numeric value expression.

CEILING is a Trafodion SQL extension.

```
CEILING (numeric-expression)
```

numeric-expression

is an SQL numeric value expression that specifies the value for the argument of the CEILING function. See "Numeric Value Expressions" (page 218).

Example of CEILING

This function returns the integer value 3.00000000000000000E+000, represented as a FLOAT data type:

```
CEILING (2.25)
```

CHAR Function

The CHAR function returns the character that has the specified code value, which must be of exact numeric with scale 0.

CHAR is a Trafodion SQL extension.

```
CHAR(code-value, [,char-set-name])
```

code-value

is a valid code value in the character set in use.

char-set-name

can be ISO88591 or UTF8. The returned character will be associated with the character set specified by *char-set-name*.

The default for *char-set-name* is ISO88591.

Considerations for CHAR

- For the ISO88591 character set, the return type is VARCHAR(1).
- For the UTF8 character set, the return type is VARCHAR(1).

Example of CHAR

Select the column CUSTNAME and return the ASCII code of the first character of the customer name and its CHAR value:

```
SELECT custname, ASCII (custname), CHAR (ASCII (custname))
FROM sales.customer;

CUSTNAME            (EXPR)   (EXPR)
------------------  -------  -------
CENTRAL UNIVERSITY  67       C
BROWN MEDICAL CO    66       B
STEVENS SUPPLY      83       S
PREMIER INSURANCE   80       P
...                 ...      ...

--- 15 row(s) selected.
```

CHAR_LENGTH Function

The CHAR_LENGTH function returns the number of characters in a string. You can also use CHARACTER_LENGTH. Every character, including multibyte characters, counts as one character.

```
CHAR[ACTER]_LENGTH (string-value-expression)
```

string-value-expression

specifies the string value expression for which to return the length in characters. Trafodion SQL returns the result as a two-byte signed integer with a scale of zero. If *string-value-expression* is null, Trafodion SQL returns a length of null. See "Character Value Expressions" (page 211).

Considerations for CHAR_LENGTH

CHAR and VARCHAR Operands

For a column declared as fixed CHAR, Trafodion SQL returns the maximum length of that column. For a VARCHAR column, Trafodion SQL returns the actual length of the string stored in that column.

Examples of CHAR_LENGTH

- This function returns 12 as the result. The concatenation operator is denoted by two vertical bars (||).

  ```
  CHAR_LENGTH ('ROBERT' || ' ' || 'SMITH')
  ```

- The string '' is the null (or empty) string. This function returns 0 (zero):

  ```
  CHAR_LENGTH ('')
  ```

- The DEPTNAME column has data type CHAR(12). Therefore, this function always returns 12:

  ```
  CHAR_LENGTH (deptname)
  ```

- The PROJDESC column in the PROJECT table has data type VARCHAR(18). This function returns the actual length of the column value—not 18 for shorter strings—because it is a VARCHAR value:

  ```
  SELECT CHAR_LENGTH (projdesc)
  FROM persnl.project;

  (EXPR)
  ----------

          14
          13
          13
          17
           9
           9
  --- 6 row(s) selected.
  ```

COALESCE Function

The COALESCE function returns the value of the first expression in the list that does not have a NULL value or if all the expressions have NULL values, the function returns a NULL value.

```
COALESCE (expr1, expr2, ...)
```

expr1

an expression to be compared.

expr2

an expression to be compared.

Example of COALESCE

COALESCE returns the value of the first operand that is not NULL:

```
SELECT COALESCE (office_phone, cell_phone, home_phone, pager,
    fax_num, '411') from emptbl;
```

CODE_VALUE Function

The CODE_VALUE function returns an unsigned integer (INTEGER UNSIGNED) that is the code point of the first character in a character value expression that can be associated with one of the supported character sets.

CODE_VALUE is a Trafodion SQL extension.

```
CODE_VALUE(character-value-expression)
_character-set
```

character-value-expression

is a character string.

Example of CODE_VALUE Function

This function returns 97 as the result:

```
>>select code_value('abc') from (values(1))x;

(EXPR)
----------

        97
```

CONCAT Function

The CONCAT function returns the concatenation of two character value expressions as a character string value. You can also use the concatenation operator (||).

CONCAT is a Trafodion SQL extension.

```
CONCAT (character-expr-1, character-expr-2)
```

character-expr-1, *character-expr-2*

are SQL character value expressions (of data type CHAR or VARCHAR) that specify two strings of characters. Both character value expressions must be either ISO8859-1 character expressions or UTF8 character expressions. The result of the CONCAT function is the concatenation of *character-expr-1* with *character-expr-2*. The result type is CHAR if both expressions are of type CHAR and it is VARCHAR if either of the expressions is of type VARCHAR. See "Character Value Expressions" (page 211).

Concatenation Operator (||)

The concatenation operator, denoted by two vertical bars (||), concatenates two string values to form a new string value. To indicate that two strings are concatenated, connect the strings with two vertical bars (||):

```
character-expr-1 || character-expr-2
```

An operand can be any SQL value expression of data type CHAR or VARCHAR.

Considerations for CONCAT

Operands

A string value can be specified by any character value expression, such as a character string literal, character string function, column reference, aggregate function, scalar subquery, CASE expression, or CAST expression. The value of the operand must be of type CHAR or VARCHAR.

If you use the CAST expression, you must specify the length of CHAR or VARCHAR.

SQL Parameters

You can concatenate an SQL parameter and a character value expression. The concatenated parameter takes on the data type attributes of the character value expression. Consider this example, where `?p` is assigned a string value of `'5 March'`:

```
?p || ' 2002'
```

The type assignment of the parameter `?p` becomes CHAR(5), the same data type as the character literal `' 2002'`. Because you assigned a string value of more than five characters to `?p`, Trafodion SQL returns a truncation warning, and the result of the concatenation is `5 Mar 2002`.

To specify the type assignment of the parameter, use the CAST expression on the parameter as:

```
CAST(?p AS CHAR(7)) || '2002'
```

In this example, the parameter is not truncated, and the result of the concatenation is `5 March 2002`.

Examples of CONCAT

- Insert information consisting of a single character string. Use the CONCAT function to construct and insert the value:

  ```
  INSERT INTO stats.logfile
  (user_key, user_info)
  VALUES (001, CONCAT ('Executed at ',
          CAST (CURRENT_TIMESTAMP AS CHAR(26))));
  ```

- Use the concatenation operator || to construct and insert the value:

```
INSERT INTO stats.logfile
(user_key, user_info)
VALUES (002, 'Executed at ' ||
        CAST (CURRENT_TIMESTAMP AS CHAR(26)));
```

CONVERTTOHEX Function

The CONVERTTOHEX function converts the specified value expression to hexadecimal for display purposes.

CONVERTTOHEX is a Trafodion SQL extension.

```
CONVERTTOHEX (expression)
```

expression

is any numeric, character, datetime, or interval expression.

The primary purpose of the CONVERTTOHEX function is to eliminate any doubt as to the exact value in a column. It is particularly useful for character expressions where some characters may be from character sets that are not supported by the client terminal's locale or may be control codes or other non-displayable characters.

Considerations for CONVERTTOHEX

Although CONVERTTOHEX is usable on datetime and interval expressions, the displayed output shows the internal value and is, consequently, not particularly meaningful to general users and is subject to change in future releases.

CONVERTTOHEX returns ASCII characters in ISO8859-1 encoding.

Examples of CONVERTTOHEX

- Display the contents of a smallint, integer, and largeint in hexadecimal:

```
CREATE  TABLE   EG (S1 smallint,  I1 int,  L1 largeint);
INSERT  INTO  EG  VALUES( 37, 2147483647, 2305843009213693951);
SELECT CONVERTTOHEX(S1), CONVERTTOHEX(I1), CONVERTTOHEX(L1) from EG;

(EXPR)   (EXPR)       (EXPR)
------   --------     ----------------
0025     7FFFFFFF     1FFFFFFFFFFFFFFF
```

- Display the contents of a CHAR(4) column, a VARCHAR(4) column, and a CHAR(4) column that uses the UTF8 character set. The varchar column does not have a trailing space character as the fixed-length columns have:

```
CREATE TABLE EG_CH (FC4 CHAR(4), VC4 VARCHAR(4), FC4U CHAR(4) CHARACTER SET UTF8);
INSERT INTO EG_CH values('ABC', 'abc', _UTF8'abc');
SELECT CONVERTTOHEX(FC4), CONVERTTOHEX(VC4), CONVERTTOHEX(FC4U) from EG_CH;

(EXPR)      (EXPR)     (EXPR)
--------    --------   ----------------

41424320    616263     0061006200630020
```

- Display the internal values for a DATE column, a TIME column, a TIMESTAMP(2) column, and a TIMESTAMP(6) column:

```
CREATE TABLE DT (D1 date, T1 time, TS1 timestamp(2), TS2 timestamp(6) );
INSERT INTO DT values(current_date, current_time, current_timestamp,
    current_timestamp);
SELECT CONVERTTOHEX(D1), CONVERTTOHEX(T1), CONVERTTOHEX(TS1),
    CONVERTTOHEX(TS2) from DT;

(EXPR)        (EXPR)      (EXPR)                      EXPR)
-----------   ---------   -------------------------   -------------------------
07D8040F      0E201E      07D8040F0E201E00000035      07D8040F0E201E00081ABB
```

- Display the internal values for an INTERVAL YEAR column, an INTERVAL YEAR(2) TO MONTH column, and an INTERVAL DAY TO SECOND column:

```
CREATE TABLE IVT ( IV1 interval year, IV2 interval year(2) to month,
    IV3 interval day to second);
INSERT INTO IVT values( interval '1' year, interval '3-2' year(2) to month,
```

```
    interval '31:14:59:58' day to second);
 SELECT CONVERTTOHEX(IV1), CONVERTTOHEX(IV2), CONVERTTOHEX(IV3) from IVT;

(EXPR)     (EXPR)         (EXPR)
------     --------       ----------------------

0001       0026           0000027C2F9CB780
```

CONVERTTIMESTAMP Function

The CONVERTTIMESTAMP function converts a Julian timestamp to a value with data type TIMESTAMP.

CONVERTTIMESTAMP is a Trafodion SQL extension.

```
CONVERTTIMESTAMP (julian-timestamp)
```

julian-timestamp

is an expression that evaluates to a Julian timestamp, which is a LARGEINT value.

Considerations for CONVERTTIMESTAMP

The *julian-timestamp* value must be in the range from 148731163200000000 to 274927348799999999.

Relationship to the JULIANTIMESTAMP Function

The operand of CONVERTTIMESTAMP is a Julian timestamp, and the function result is a value of data type TIMESTAMP. The operand of the CONVERTTIMESTAMP function is a value of data type TIMESTAMP, and the function result is a Julian timestamp. That is, the two functions have an inverse relationship to one another.

Use of CONVERTTIMESTAMP

You can use the inverse relationship between the JULIANTIMESTAMP and CONVERTTIMESTAMP functions to insert Julian timestamp columns into your database and display these column values in a TIMESTAMP format.

Examples of CONVERTTIMESTAMP

- Suppose that the EMPLOYEE table includes a column, named HIRE_DATE, which contains the hire date of each employee as a Julian timestamp. Convert the Julian timestamp into a TIMESTAMP value:

  ```
  SELECT CONVERTTIMESTAMP (hire_date)
  FROM persnl.employee;
  ```

- This example illustrates the inverse relationship between JULIANTIMESTAMP and CONVERTTIMESTAMP.

  ```
  SELECT CONVERTTIMESTAMP (JULIANTIMESTAMP (ship_timestamp))
  FROM persnl.project;
  ```

  If, for example, the value of SHIP_TIMESTAMP is 2008-04-03 21:05:36.143000, the result of CONVERTTIMESTAMP(JULIANTIMESTAMP(ship_timestamp)) is the same value, 2008-04-03 21:05:36.143000.

COS Function

The COS function returns the cosine of a numeric value expression, where the expression is an angle expressed in radians.

COS is a Trafodion SQL extension.

```
COS (numeric-expression)
```

numeric-expression

is an SQL numeric value expression that specifies the value for the argument of the COS function. See "Numeric Value Expressions" (page 218).

Example of COS

This function returns the value 9.39680940386503680E-001, or approximately 0.9397, the cosine of 0.3491 (which is 20 degrees):

```
COS (0.3491)
```

COSH Function

The COSH function returns the hyperbolic cosine of a numeric value expression, where the expression is an angle expressed in radians.

COSH is a Trafodion SQL extension.

```
COSH (numeric-expression)
```

numeric-expression

is an SQL numeric value expression that specifies the value for the argument of the COSH function. See "Numeric Value Expressions" (page 218).

Example of COSH

This function returns the value 1.88842387716101568E+000, or approximately 1.8884, the hyperbolic cosine of 1.25 in radians:

```
COSH (1.25)
```

COUNT Function

The COUNT function counts the number of rows that result from a query or the number of rows that contain a distinct value in a specific column. The result of COUNT is data type LARGEINT. The result can never be NULL.

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

COUNT (*)

returns the number of rows in the table specified in the FROM clause of the SELECT statement that contains COUNT (*). If the result table is empty (that is, no rows are returned by the query) COUNT (*) returns zero.

ALL | DISTINCT

returns the number of all rows or the number of distinct rows in the one-column table derived from the evaluation of *expression*. The default option is ALL, which causes duplicate values to be included. If you specify DISTINCT, duplicate values are eliminated before the COUNT function is applied.

expression

specifies a value expression that determines the values to count. The *expression* cannot contain an aggregate function or a subquery. The DISTINCT clause specifies that the COUNT function operates on distinct values from the one-column table derived from the evaluation of *expression*. See "Expressions" (page 211).

Considerations for COUNT

Operands of the Expression

The operand of COUNT is either * or an expression that includes columns from the result table specified by the SELECT statement that contains COUNT. However, the expression cannot include an aggregate function or a subquery. These expressions are valid:

```
COUNT (*)
COUNT (DISTINCT JOBCODE)
COUNT (UNIT_PRICE * QTY_ORDERED)
```

Nulls

COUNT is evaluated after eliminating all nulls from the one-column table specified by the operand. If the table has no rows, COUNT returns zero.

COUNT(*) does not eliminate null rows from the table specified in the FROM clause of the SELECT statement. If all rows in a table are null, COUNT(*) returns the number of rows in the table.

Examples of COUNT

- Count the number of rows in the EMPLOYEE table:

```
SELECT COUNT (*)
FROM persnl.employee;

(EXPR)
-----------
         62

--- 1 row(s) selected.
```

- Count the number of employees who have a job code in the EMPLOYEE table:

```
SELECT COUNT (jobcode)
FROM persnl.employee;

(EXPR)
-----------
```

```
         56

--- 1 row(s) selected.

SELECT COUNT(*)
FROM persnl.employee
WHERE jobcode IS NOT NULL;

(EXPR)
-----------
         56

--- 1 row(s) selected.
```

- Count the number of distinct departments in the EMPLOYEE table:

```
SELECT COUNT (DISTINCT deptnum)
FROM persnl.employee;

(EXPR)
-----------
         11

--- 1 row(s) selected.
```

CURRENT Function

The CURRENT function returns a value of type TIMESTAMP based on the current local date and time.

The function is evaluated once when the query starts execution and is not reevaluated (even if it is a long running query).

You can also use "CURRENT_TIMESTAMP Function" (page 318).

```
CURRENT [(precision)]
```

precision

is an integer value in the range 0 to 6 that specifies the precision of (the number of decimal places in) the fractional seconds in the returned value. The default is 6.

For example, the function CURRENT (2) returns the current date and time as a value of data type TIMESTAMP, where the precision of the fractional seconds is 2, for example, 2008-06-26 09:01:20.89. The value returned is not a string value.

Example of CURRENT

The PROJECT table contains a column SHIP_TIMESTAMP of data type TIMESTAMP. Update a row by using the CURRENT value:

```
UPDATE persnl.project
SET ship_timestamp = CURRENT
WHERE projcode = 1000;
```

CURRENT_DATE Function

The CURRENT_DATE function returns the local current date as a value of type DATE.

The function is evaluated once when the query starts execution and is not reevaluated (even if it is a long running query).

```
CURRENT_DATE
```

The CURRENT_DATE function returns the current date, such as 2008-09-28. The value returned is a value of type DATE, not a string value.

Examples of CURRENT_DATE

- Select rows from the ORDERS table based on the current date:

```
SELECT * FROM sales.orders
WHERE deliv_date >= CURRENT_DATE;
```

- The PROJECT table has a column EST_COMPLETE of type INTERVAL DAY. If the current date is the start date of your project, determine the estimated date of completion:

```
SELECT projdesc, CURRENT_DATE + est_complete
FROM persnl.project;

Project/Description  (EXPR)
-------------------  ----------
SALT LAKE CITY       2008-01-18
ROSS PRODUCTS        2008-02-02
MONTANA TOOLS        2008-03-03
AHAUS TOOL/SUPPLY    2008-03-03
THE WORKS            2008-02-02
THE WORKS            2008-02-02

--- 6 row(s) selected.
```

CURRENT_TIME Function

The CURRENT_TIME function returns the current local time as a value of type TIME.

The function is evaluated once when the query starts execution and is not reevaluated (even if it is a long running query).

```
CURRENT_TIME [(precision)]
```

precision

is an integer value in the range 0 to 6 that specifies the precision of (the number of decimal places in) the fractional seconds in the returned value. The default is 0.

For example, the function CURRENT_TIME (2) returns the current time as a value of data type TIME, where the precision of the fractional seconds is 2, for example, 14:01:59.30. The value returned is not a string value.

Example of CURRENT_TIME

Use CURRENT_DATE and CURRENT_TIME as a value in an inserted row:

```
INSERT INTO stats.logfile
(user_key, run_date, run_time, user_name)
VALUES (001, CURRENT_DATE, CURRENT_TIME, 'JuBrock');
```

CURRENT_TIMESTAMP Function

The CURRENT_TIMESTAMP function returns a value of type TIMESTAMP based on the current local date and time.

The function is evaluated once when the query starts execution and is not reevaluated (even if it is a long running query).

You can also use the "CURRENT Function" (page 315).

```
CURRENT_TIMESTAMP [(precision)]
```

precision

is an integer value in the range 0 to 6 that specifies the precision of (the number of decimal places in) the fractional seconds in the returned value. The default is 6.

For example, the function CURRENT_TIMESTAMP (2) returns the current date and time as a value of data type TIMESTAMP, where the precision of the fractional seconds is 2; for example, 2008-06-26 09:01:20.89. The value returned is not a string value.

Example of CURRENT_TIMESTAMP

The PROJECT table contains a column SHIP_TIMESTAMP of data type TIMESTAMP. Update a row by using the CURRENT_TIMESTAMP value:

```
UPDATE persnl.project
SET ship_timestamp = CURRENT_TIMESTAMP
WHERE projcode = 1000;
```

CURRENT_USER Function

The CURRENT_USER function returns the database username of the current user who invoked the function. The current user is the authenticated user who started the session. That database username is used for authorization of SQL statements in the current session.

```
CURRENT_USER
```

The CURRENT_USER function is similar to the "USER Function" (page 425).

Considerations for CURRENT_USER

- This function can be specified only in the top level of a SELECT statement.
- The value returned is string data type VARCHAR(128) and is in ISO8859-1 encoding.

Example of CURRENT_USER

This example retrieves the database username for the current user:

```
SELECT CURRENT_USER FROM (values(1)) x(a);

(EXPR)
----------------------
TSHAW
--- 1 row(s) selected.
```

DATE_ADD Function

The DATE_ADD function adds the interval specified by *interval_expression* to *datetime_expr*. If the specified interval is in years or months, DATE_ADD normalizes the result. See "Standard Normalization" (page 278). The type of the *datetime_expr* is returned, unless the *interval_expression* contains any time components, then a timestamp is returned.

DATE_ADD is a Trafodion SQL extension.

```
DATE_ADD (datetime_expr, interval_expression)
```

datetime_expr

is an expression that evaluates to a datetime value of type DATE or TIMESTAMP. See "Datetime Value Expressions" (page 212).

interval_expression

is an expression that can be combined in specific ways with addition operators. The *interval_expression* accepts all interval expression types that the Trafodion database software considers as valid interval expressions. See "Interval Value Expressions" (page 215).

Examples of DATE_ADD

- This function returns the value DATE '2007-03-07'

  ```
  DATE_ADD(DATE '2007-02-28', INTERVAL '7' DAY)
  ```

- This function returns the value DATE '2008-03-06'

  ```
  DATE_ADD(DATE '2008-02-28', INTERVAL '7' DAY)
  ```

- This function returns the timestamp '2008-03-07 00:00:00'

  ```
  DATE_ADD(timestamp'2008-02-29 00:00:00', INTERVAL '7' DAY)
  ```

- This function returns the timestamp '2008-02-28 23:59:59'

  ```
  DATE_ADD(timestamp '2007-02-28 23:59:59', INTERVAL '12' MONTH)
  ```

Note: compare the last example with the last example under DATE_SUB.

DATE_SUB Function

The DATE_SUB function subtracts the specified *interval_expression* from *datetime_expr*. If the specified interval is in years or months, DATE_SUB normalizes the result. See "Standard Normalization" (page 278). The type of the *datetime_expr* is returned, unless the *interval_expression* contains any time components, then a timestamp is returned.

DATE_SUB is a Trafodion SQL extension.

```
DATE_SUB (datetime_expr, interval_expression)
```

datetime_expr

is an expression that evaluates to a datetime value of type DATE or TIMESTAMP. See "Datetime Value Expressions" (page 212).

interval_expression

is an expression that can be combined in specific ways with subtraction operators. The *interval_expression* accepts all interval expression types that the Trafodion database software considers as valid interval expressions. See "Interval Value Expressions" (page 215).

Examples of DATE_SUB

- This function returns the value DATE '2009-02-28'

```
DATE_SUB(DATE '2009-03-07', INTERVAL'7' DAY)
```

- This function returns the value DATE '2008-02-29'

```
DATE_SUB(DATE '2008-03-07', INTERVAL'7' DAY)
```

- This function returns the timestamp '2008-02-29 00:00:00'

```
DATE_SUB(timestamp '2008-03-31 00:00:00', INTERVAL '31' DAY)
```

- This function returns the timestamp '2007-02-28 23:59:59'

```
DATE_SUB(timestamp '2008-02-29 23:59:59', INTERVAL '12' MONTH)
```

DATEADD Function

The DATEADD function adds the interval of time specified by *datepart* and *num_expr* to *datetime_expr*. If the specified interval is in years or months, DATEADD normalizes the result. See "Standard Normalization" (page 278). The type of the *datetime_expr* is returned, unless the interval expression contains any time components, then a timestamp is returned.

DATEADD is a Trafodion SQL extension.

```
DATEADD(datepart, num_expr, datetime_expr)
```

datepart

is YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, QUARTER, WEEK, or one of the following abbreviations:

YEAR	*YY* and *YYYY*
MONTH	*M* and *MM*
DAY	*D* and *DD*
HOUR	*HH*
MINUTE	*MI* and *M*
SECOND	*SS* and *S*
QUARTER	*Q* and *QQ*
WEEK	*WW* and *WK*

num_expr

is an SQL exact numeric value expression that specifies how many *datepart* units of time are to be added to *datetime_expr*. If *num_expr* has a fractional portion, it is ignored. If *num_expr* is negative, the return value precedes *datetime_expr* by the specified amount of time. See "Numeric Value Expressions" (page 218).

datetime_expr

is an expression that evaluates to a datetime value of type DATE or TIMESTAMP. The type of the *datetime_expression* is returned, unless the interval expression contains any time components, then a timestamp is returned. See "Datetime Value Expressions" (page 212).

Examples of DATEADD

- This function adds seven days to the date specified in *start_date*

```
DATEADD(DAY, 7,start_date)
```

- This function returns the value DATE '2009-03-07'

```
DATEADD(DAY, 7 , DATE '2009-02-28')
```

- This function returns the value DATE '2008-03-06'

```
DATEADD(DAY, 7, DATE '2008-02-28')
```

- This function returns the timestamp '2008-03-07 00:00:00'

```
DATEADD(DAY, 7, timestamp'2008-02-29 00:00:00')
```

DATEDIFF Function

The DATEDIFF function returns the integer value for the number of *datepart* units of time between *startdate* and *enddate*. If *enddate* precedes *startdate*, the return value is negative or zero.

DATEDIFF is a Trafodion SQL extension.

```
DATEDIFF (datepart, startdate, enddate)
```

datepart

is YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, QUARTER, WEEK, or one of the following abbreviations

YEAR	*YY* and *YYYY*
MONTH	*M* and *MM*
DAY	*D* and *DD*
HOUR	*HH*
MINUTE	*MI* and *M*
SECOND	*SS* and *S*
QUARTER	*Q* and QQ
WEEK	*WW* and *WK*

startdate

may be of type DATE or TIMESTAMP. See "Datetime Value Expressions" (page 212).

enddate

may be of type DATE or TIMESTAMP. See "Datetime Value Expressions" (page 212).

The method of counting crossed boundaries such as days, minutes, and seconds makes the result given by DATEDIFF consistent across all data types. The result is a signed integer value equal to the number of datepart boundaries crossed between the first and second date.

For example, the number of weeks between Sunday, January 4, and Sunday, January 11, is 1. The number of months between March 31 and April 1 would be 1 because the month boundary is crossed from March to April. The DATEDIFF function generates an error if the result is out of range for integer values. For seconds, the maximum number is equivalent to approximately 68 years. The DATEDIFF function generates an error if a difference in weeks is requested and one of the two dates precedes January 7 of the year 0001.

Examples of DATEDIFF

- This function returns the value of 0 because no one-second boundaries are crossed.

```
DATEDIFF(SECOND, TIMESTAMP '2006-09-12 11:59:58.999998', TIMESTAMP '2006-09-12 11:59:58.999999')
```

- This function returns the value 1 because a one-second boundary is crossed even though the two timestamps differ by only one microsecond.

```
DATEDIFF(SECOND, TIMESTAMP '2006-09-12 11:59:58.999999', TIMESTAMP '2006-09-12 11:59:59.000000')
```

- This function returns the value of 0.

```
DATEDIFF(YEAR, TIMESTAMP '2006-12-31 23:59:59.999998', TIMESTAMP '2006-12-31 23:59:59.999999')
```

- This function returns the value of 1 because a year boundary is crossed.

```
DATEDIFF(YEAR, TIMESTAMP '2006-12-31 23:59:59.999999', TIMESTAMP '2007-01-01 00:00:00.000000')
```

- This function returns the value of 2 because two WEEK boundaries are crossed.

```
DATEDIFF(WEEK, DATE '2006-01-01', DATE '2006-01-09')
```

- This function returns the value of -29.

```
DATEDIFF(DAY, DATE '2008-03-01', DATE '2008-02-01')
```

DATEFORMAT Function

The DATEFORMAT function returns a datetime value as a character string literal in the DEFAULT, USA, or EUROPEAN format. The data type of the result is CHAR.

DATEFORMAT is a Trafodion SQL extension.

```
DATEFORMAT (datetime-expression,{DEFAULT | USA | EUROPEAN})
```

datetime-expression

is an expression that evaluates to a datetime value of type DATE, TIME, or TIMESTAMP. See "Datetime Value Expressions" (page 212).

DEFAULT | USA | EUROPEAN

specifies a format for a datetime value. See "Datetime Literals" (page 226).

Considerations for DATEFORMAT

The DATEFORMAT function returns the datetime value in ISO8859-1 encoding.

Examples of DATEFORMAT

- Convert a datetime literal in DEFAULT format to a string in USA format:

  ```
  DATEFORMAT (TIMESTAMP '2008-06-20 14:20:20.00', USA)
  ```

  The function returns this string literal:

  ```
  '06/20/2008 02:20:20.00 PM'
  ```

- Convert a datetime literal in DEFAULT format to a string in European format:

  ```
  DATEFORMAT (TIMESTAMP '2008-06-20 14:20:20.00', EUROPEAN)
  ```

  The function returns this string literal:

  ```
  '20.06.2008 14.20.20.00'
  ```

DATE_PART Function (of an Interval)

The DATE_PART function extracts the datetime field specified by *text* from the *interval* value specified by *interval* and returns the result as an exact numeric value. The DATE_PART function accepts the specification of 'YEAR', 'MONTH', 'DAY', 'HOUR', 'MINUTE', or 'SECOND' for text.

DATE_PART is a Trafodion SQL extension.

```
DATE_PART (text, interval)
```

text

specifies YEAR, MONTH, DAY, HOUR, MINUTE, or SECOND. The value must be enclosed in single quotes.

interval

interval accepts all interval expression types that the Trafodion database software considers as valid interval expressions. See "Interval Value Expressions" (page 215).

The DATE_PART(*text*, *interval*) is equivalent to EXTRACT(*text*, *interval*), except that the DATE_PART function requires single quotes around the text specification, where EXTRACT does not allow single quotes.

When SECOND is specified the fractional part of the second is returned.

Examples of DATE_PART

- This function returns the value of 7.

  ```
  DATE_PART('DAY', INTERVAL '07:04' DAY TO HOUR)
  ```

- This function returns the value of 6.

  ```
  DATE_PART('MONTH', INTERVAL '6' MONTH)
  ```

- This function returns the value of 36.33.

  ```
  DATE_PART('SECOND', INTERVAL '5:2:15:36.33' DAY TO SECOND(2))
  ```

DATE_PART Function (of a Timestamp)

The DATE_PART function extracts the datetime field specified by *text* from the datetime value specified by *datetime_expr* and returns the result as an exact numeric value. The DATE_PART function accepts the specification of 'YEAR', 'YEARQUARTER', 'YEARMONTH', 'YEARWEEK', 'MONTH', 'DAY', 'HOUR', 'MINUTE', or 'SECOND' for text.

The DATE_PART function of a timestamp can be changed to DATE_PART function of a datetime because the second argument can be either a timestamp or a date expression.

DATE_PART is a Trafodion extension.

```
DATE_PART(text, datetime_expr)
```

text

specifies YEAR, YEARQUARTER, YEARMONTH, YEARWEEK, MONTH, DAY, HOUR, MINUTE, or SECOND. The value must be enclosed in single quotes.

- YEARMONTH: Extracts the year and the month, as a 6-digit integer of the form `yyyymm (100 * year + month)`.
- YEARQUARTER: Extracts the year and quarter, as a 5-digit integer of the form `yyyyq`, `(10 * year + quarter)` with `q` being `1` for the first quarter, `2` for the second, and so on.
- YEARWEEK: Extracts the year and week of the year, as a 6-digit integer of the form `yyyyww (100 * year + week)`. The week number will be computed in the same way as in the WEEK function.

datetime_expr

is an expression that evaluates to a datetime value of type DATE or TIMESTAMP. See "Datetime Value Expressions" (page 212).

DATE_PART(*text*, *datetime_expr*) is mostly equivalent to EXTRACT(*text*, *datetime_expr*), except that DATE_PART requires single quotes around the text specification where EXTRACT does not allow single quotes. In addition, you cannot use the YEARQUARTER, YEARMONTH, and YEARWEEK text specification with EXTRACT.

Examples of DATE_PART

- This function returns the value of 12.

  ```
  DATE_PART('month', date'12/05/2006')
  ```

- This function returns the value of 2006.

  ```
  DATE_PART('year', date'12/05/2006')
  ```

- This function returns the value of 31.

  ```
  DATE_PART('day', TIMESTAMP '2006-12-31 11:59:59.999999')
  ```

- This function returns the value 201107.

  ```
  DATE_PART('YEARMONTH', date '2011-07-25')
  ```

DATE_TRUNC Function

The DATE_TRUNC function returns a value of type TIMESTAMP, which has all fields of lesser precision than *text* set to zero (or 1 in the case of months or days).

DATE_TRUNC is a Trafodion SQL extension.

```
DATE_TRUNC(text, datetime_expr)
```

text

specifies 'YEAR', 'MONTH', 'DAY', 'HOUR', 'MINUTE', or 'SECOND'. The DATE_TRUNC function also accepts the specification of 'CENTURY' or 'DECADE'.

datetime_expr

is an expression that evaluates to a datetime value of type DATE or TIMESTAMP. DATE_TRUNC returns a value of type TIMESTAMP which has all fields of lesser precision than *text* set to zero (or 1 in the case of months or days). See "Datetime Value Expressions" (page 212).

Examples of DATE_TRUNC

- This function returns the value of TIMESTAMP '2006-12-31 00:00:00'.

  ```
  DATE_TRUNC('DAY', TIMESTAMP '2006-12-31 11:59:59')
  ```

- This function returns the value of TIMESTAMP '2006-01-01 00:00:00'

  ```
  DATE_TRUNC('YEAR', TIMESTAMP '2006-12-31 11:59:59')
  ```

- This function returns the value of TIMESTAMP '2006-12-01 00:00:00'

  ```
  DATE_TRUNC('MONTH', DATE '2006-12-31')
  ```

- Restrictions:
  - DATE_TRUNC('DECADE', ...) cannot be used on years less than 10.
  - DATE_TRUNC('CENTURY', ...) cannot be used on years less than 100.

DAY Function

The DAY function converts a DATE or TIMESTAMP expression into an INTEGER value in the range 1 through 31 that represents the corresponding day of the month. The result returned by the DAY function is equal to the result returned by the DAYOFMONTH function.

DAY is a Trafodion SQL extension.

```
DAY (datetime-expression)
```

datetime-expression

is an expression that evaluates to a datetime value of type DATE or TIMESTAMP. See "Datetime Value Expressions" (page 212).

Example of DAY

Return an integer that represents the day of the month from the START_DATE column of the PROJECT table:

```
SELECT start_date, ship_timestamp, DAY(start_date)
FROM persnl.project
WHERE projcode = 1000;

Start/Date  Time/Shipped                 (EXPR)
----------  --------------------------  ------
2008-04-10  2008-04-21 08:15:00.000000      10
```

DAYNAME Function

The DAYNAME function converts a DATE or TIMESTAMP expression into a character literal that is the name of the day of the week (Sunday, Monday, and so on).

DAYNAME is a Trafodion SQL extension.

```
DAYNAME (datetime-expression)
```

datetime-expression

is an expression that evaluates to a datetime value of type DATE or TIMESTAMP. See "Datetime Value Expressions" (page 212).

Considerations for DAYNAME

The DAYNAME function returns the name of the day in ISO8859-1.

Example of DAYNAME

Return the name of the day of the week from the START_DATE column in the PROJECT table:

```
SELECT start_date, ship_timestamp, DAYNAME(start_date)
FROM persnl.project
WHERE projcode = 1000;

Start/Date  Time/Shipped                 (EXPR)
----------  --------------------------  ---------
2008-04-10  2008-04-21 08:15:00.000000  Thursday
```

DAYOFMONTH Function

The DAYOFMONTH function converts a DATE or TIMESTAMP expression into an INTEGER value in the range 1 through 31 that represents the corresponding day of the month. The result returned by the DAYOFMONTH function is equal to the result returned by the DAY function.

DAYOFMONTH is a Trafodion SQL extension.

```
DAYOFMONTH (datetime-expression)
```

datetime-expression

is an expression that evaluates to a datetime value of type DATE or TIMESTAMP. See "Datetime Value Expressions" (page 212).

Examples of DAYOFMONTH

Return an integer that represents the day of the month from the START_DATE column of the PROJECT table:

```
SELECT start_date, ship_timestamp, DAYOFMONTH(start_date)
FROM persnl.project
WHERE projcode = 1000;

Start/Date  Time/Shipped                 (EXPR)
----------  --------------------------  ------
2008-04-10  2008-04-21 08:15:00.000000      10
```

DAYOFWEEK Function

The DAYOFWEEK function converts a DATE or TIMESTAMP expression into an INTEGER value in the range 1 through 7 that represents the corresponding day of the week. The value 1 represents Sunday, 2 represents Monday, and so forth.

DAYOFWEEK is a Trafodion SQL extension.

```
DAYOFWEEK (datetime-expression)
```

datetime-expression

is an expression that evaluates to a datetime value of type DATE or TIMESTAMP. See "Datetime Value Expressions" (page 212).

Example of DAYOFWEEK

Return an integer that represents the day of the week from the START_DATE column in the PROJECT table:

```
SELECT start_date, ship_timestamp, DAYOFWEEK(start_date)
FROM persnl.project
WHERE projcode = 1000;

Start/Date  Time/Shipped                 (EXPR)
----------  --------------------------  ------
2008-04-10  2008-04-21 08:15:00.000000       5
```

The value returned is 5, representing Thursday. The week begins on Sunday.

DAYOFYEAR Function

The DAYOFYEAR function converts a DATE or TIMESTAMP expression into an INTEGER value in the range 1 through 366 that represents the corresponding day of the year.

DAYOFYEAR is a Trafodion SQL extension.

```
DAYOFYEAR (datetime-expression)
```

datetime-expression

is an expression that evaluates to a datetime value of type DATE or TIMESTAMP. See "Datetime Value Expressions" (page 212).

Example of DAYOFYEAR

Return an integer that represents the day of the year from the START_DATE column in the PROJECT table:

```
SELECT start_date, ship_timestamp, DAYOFYEAR(start_date)
FROM persnl.project
WHERE projcode = 1000;

Start/Date  Time/Shipped                 (EXPR)
----------  --------------------------  --------------------
2008-04-10  2008-04-21 08:15:00.000000                   101
```

DECODE Function

The DECODE function compares *expr* to each *test_expr* value one by one in the order provided. If *expr* is equal to a *test_expr*, then the corresponding *retval* is returned. If no match is found, *default* is returned. If no match is found and *default* is omitted, NULL is returned.

DECODE is a Trafodion SQL extension.

```
DECODE (expr, test_expr, retval [, test_expr2, retval2 ... ] [ , default ] )
```

expr
is an SQL expression.

test_expr, *test_expr*,..
are each an SQL expression of a type comparable to that of *expr*.

retval
is an SQL expression.

default, *retval2*,..
are each an SQL expression of a type comparable to that of *retval*.

Considerations for DECODE

In a DECODE function, two nulls are considered to be equivalent. If *expr* is null, then the returned value is the *retval* of the first *test_expr* that is also null.

The *expr*, *test_expr*, *retval*, and *default* values can be derived from expressions.

The arguments can be any of the numeric types or character types. However, *expr* and each *test_expr* value must be of comparable types. If *expr* and *test_expr* values are character types, they must be in the same character set (to be comparable types.)

All the *retval* values and the *default* value, if any, must be of comparable types.

If *expr* and a *test_expr* value are character data, the comparison is made using nonpadded comparison semantics.

If *expr* and a *test_expr* value are numeric data, the comparison is made with a temporary copy of one of the numbers, according to defined rules of conversion. For example, if one number is INTEGER and the other is DECIMAL, the comparison is made with a temporary copy of the integer converted to a decimal.

If all the possible return values are of fixed-length character types, the returned value is a fixed-length character string with size equal to the maximum size of all the possible return value types.

If any of the possible return values is a variable-length character type, the returned value is a variable-length character string with maximum size of all the possible return value types.

If all the possible return values are of integer types, the returned value is the same type as the largest integer type of all the possible return values.

If the returned value is of type FLOAT, the precision is the maximum precision of all the possible return values.

If all the possible returned values are of the same non-integer, numeric type (REAL, FLOAT, DOUBLE PRECISION, NUMERIC, or DECIMAL), the returned value is of that same type.

If all the possible return values are of numeric types but not all the same, and at least one is REAL, FLOAT, or DOUBLE PRECISION, then the returned value is of type DOUBLE PRECISION.

If all the possible return values are of numeric types but not all the same, none are REAL, FLOAT, or DOUBLE PRECISION, and at least one is of type NUMERIC, then the returned value is of type NUMERIC.

If all the possible return values are of numeric types, none are NUMERIC, REAL, FLOAT, or DOUBLE PRECISION, and at least one is of type DECIMAL, then the returned value will be of type DECIMAL.

If the returned value is of type NUMERIC or DECIMAL, it has a precision equal to the sum of:

- The maximum scale of all the possible return value types and
- The maximum value of (precision - scale) for all the possible return value types.

However, the precision will not exceed 18.

The scale of the returned value is the minimum of:

- the maximum scale of all the possible return value types and
- 18 - (the maximum value of (precision - scale) for all the possible return value types).

The number of components in the DECODE function, including *expr*, *test_exprs*, *retvals*, and *default*, has no limit other than the general limit of how big an SQL expression can be. However, large lists do not perform well.

The syntax `DECODE (expr, test_expr, retval [, test_expr2, retval2 ... ] [ , default ] ):`

is logically equivalent to the following:

```
    CASE WHEN (expr IS NULL AND test_expr IS NULL) OR
              expr = test_expr  THEN retval
      WHEN (expr IS NULL AND test_expr2 IS NULL) OR
            expr = test_expr2 THEN retval2
      ...
      ELSE default /* or  ELSE NULL if default not
            specified */
END
```

No special conversion of *expr*, *test_exprN*, or *retvalN* exist other than what a CASE statement normally does.

Examples of DECODE

- Example of the DECODE function:

```
SELECT emp_name,
decode(CAST (( yrs_of_service + 3) / 4 AS INT ) ,
       0,0.04,
       1,0.04,
       0.06) as perc_value
FROM employees;
SELECT supplier_name,
decode(supplier_id,    10000,   'Company A',
                       10001,   'Company B',
                       10002,   'Company C',
                       'Company D') as result
FROM suppliers;
```

- This example shows a different way of handling NULL specified as default and not specified as default explicitly:

```
SELECT decode( (?p1 || ?p2), trim(?p1), 'Hi', ?p3, null )
    from emp;
..
*** ERROR[4049] A CASE expression cannot have a result data type of both CHAR(2)
and NUMERIC(18,6).
*** ERROR[4062] The preceding error actually occurred in function
DECODE((?P1 || ?P2),(' ' TRIM ?P1), 'Hi', ?P3, NULL)
*** ERROR[8822] The statement was not prepared.
```

The last *ret-val* is an explicit NULL. When Trafodion SQL encounters this situation, it assumes that the return value will be NUMERIC(18,6). Once Trafodion SQL determines that the return values are numeric, it determines that all possible return values must be numeric. When `'Hi'` is encountered in a *ret-val* position, the error is produced because the CHAR(2) type argument is not comparable with a NUMERIC(18,6) type return value.

This statement is equivalent and will not produce an error:

```
SELECT decode( (?p1 || ?p2), trim(?p1), 'Hi' ) from emp;
```

DEGREES Function

The DEGREES function converts a numeric value expression expressed in radians to the number of degrees.

DEGREES is a Trafodion SQL extension.

```
DEGREES (numeric-expression)
```

numeric-expression

is an SQL numeric value expression that specifies the value for the argument of the DEGREES function. See "Numeric Value Expressions" (page 218).

Examples of DEGREES

- This function returns the value 45.0001059971939008 in degrees:

  ```
  DEGREES (0.78540)
  ```

- This function returns the value of 45. The function DEGREES is the inverse of the function RADIANS.

  ```
  DEGREES (RADIANS (45))
  ```

DIFF1 Function

- "Considerations for DIFF1"
- "Examples of DIFF1"

The DIFF1 function is a sequence function that calculates the amount of change in an expression from row to row in an intermediate result table ordered by a SEQUENCE BY clause in a SELECT statement. See "SEQUENCE BY Clause" (page 268).

DIFF1 is a Trafodion SQL extension.

```
DIFF1 (column-expression-a [,column-expression-b])
```

column-expression-a

specifies a derived column determined by the evaluation of the column expression. If you specify only one column as an argument, DIFF1 returns the difference between the value of the column in the current row and its value in the previous row; this version calculates the unit change in the value from row to row.

column-expression-b

specifies a derived column determined by the evaluation of the column expression. If you specify two columns as arguments, DIFF1 returns the difference in consecutive values in *column-expression-a* divided by the difference in consecutive values in *column-expression-b*.

The purpose of the second argument is to distribute the amount of change from row to row evenly over some unit of change (usually time) in another column.

Considerations for DIFF1

Equivalent Result

If you specify one argument, the result of DIFF1 is equivalent to:

```
column-expression-a - OFFSET(column-expression-a, 1)
```

If you specify two arguments, the result of DIFF1 is equivalent to:

```
DIFF1(column-expression-a) / DIFF1(column-expression-b)
```

The two-argument version involves division by the result of the DIFF1 function. To avoid divide-by-zero errors, be sure that *column-expression-b* does not contain any duplicate values whose DIFF1 computation could result in a divisor of zero.

Datetime Arguments

In general, Trafodion SQL does not allow division by a value of INTERVAL data type. However, to permit use of the two-argument version of DIFF1 with times and dates, Trafodion SQL relaxes this restriction and allows division by a value of INTERVAL data type.

Examples of DIFF1

- Retrieve the difference between the I1 column in the current row and the I1 column in the previous row:

```
SELECT DIFF1 (I1) AS DIFF1_I1
FROM mining.seqfcn
SEQUENCE BY TS;

DIFF1_I1
------------
           ?
       21959
       -9116
      -14461
        7369
```

```
--- 5 row(s) selected.
```

The first row retrieved displays null because the offset from the current row does not fall within the results set.

- Retrieve the difference between the TS column in the current row and the TS column in the previous row:

```
SELECT DIFF1 (TS) AS DIFF1_TS
FROM mining.seqfcn
SEQUENCE BY TS;

DIFF1_TS
--------------------
                    ?
     30002620.000000
    134157861.000000
    168588029.000000
    114055223.000000

--- 5 row(s) selected.
```

The results are expressed as the number of seconds. For example, the difference between TIMESTAMP '1951-02-15 14:35:49' and TIMESTAMP '1950-03-05 08:32:09' is approximately 347 days. The difference between TIMESTAMP '1955-05-18 08:40:10' and TIMESTAMP '1951-02-15 14:35:49' is approximately 4 years and 3 months, and so on.

- This query retrieves the difference in consecutive values in I1 divided by the difference in consecutive values in TS:

```
SELECT DIFF1 (I1,TS) AS DIFF1_I1TS
FROM mining.seqfcn
SEQUENCE BY TS;

DIFF1_I1TS
-------------------
                   ?
           .0007319
          -.0000679
          -.0000857
           .0000646

--- 5 row(s) selected.
```

The results are equivalent to the quotient of the results from the two preceding examples. For example, in the second row of the output of this example, 0.0007319 is equal to 21959 divided by 30002620.

DIFF2 Function

- "Considerations for DIFF2"
- "Examples of DIFF2"

The DIFF2 function is a sequence function that calculates the amount of change in a DIFF1 value from row to row in an intermediate result table ordered by a SEQUENCE BY clause in a SELECT statement. See "SEQUENCE BY Clause" (page 268).

DIFF2 is a Trafodion SQL extension.

```
DIFF2 (column-expression-a [,column-expression-b])
```

column-expression-a

specifies a derived column determined by the evaluation of the column expression. If you specify only one column as an argument, DIFF2 returns the difference between the value of DIFF1(*column-expression-a*) in the current row and the same result in the previous row.

column-expression-b

specifies a derived column determined by the evaluation of the column expression. If you specify two columns as arguments, DIFF2 returns the difference in consecutive values of DIFF1(*column-expression-a*) divided by the difference in consecutive values in *column-expression-b*.

See "DIFF1 Function" (page 337).

Considerations for DIFF2

Equivalent Result

If you specify one argument, the result of DIFF2 is equivalent to:

```
DIFF1(column-expression-a)- OFFSET(DIFF1(column-expression-a),1)
```

If you specify two arguments, the result of DIFF2 is equivalent to:

```
DIFF2(column-expression-a) / DIFF1(column-expression-b)
```

The two-argument version involves division by the result of the DIFF1 function. To avoid divide-by-zero errors, be sure that *column-expression-b* does not contain any duplicate values whose DIFF1 computation could result in a divisor of zero.

Datetime Arguments

In general, Trafodion SQL does not allow division by a value of INTERVAL data type. However, to permit use of the two-argument version of DIFF2 with times and dates, Trafodion SQL relaxes this restriction and allows division by a value of INTERVAL data type.

Examples of DIFF2

- Retrieve the difference between the value of DIFF1(I1) in the current row and the same result in the previous row:

```
SELECT DIFF2 (I1) AS DIFF2_I1
FROM mining.seqfcn
SEQUENCE BY TS;

DIFF2_I1
--------------------
                    ?
                    ?
               -31075
                -5345
                21830

--- 5 row(s) selected.
```

The results are equal to the difference of DIFF1(I1) for the current row and DIFF1(I1) of the previous row. For example, in the third row of the output of this example, -31075 is equal to -9116 minus 21959. The value -9116 is the result of DIFF1(I1) for the current row, and the value 21959 is the result of DIFF1(I1) for the previous row. See "Examples of DIFF1" (page 337).

- Retrieve the difference in consecutive values of DIFF1(I1) divided by the difference in consecutive values of TS:

```
SELECT DIFF2 (I1,TS) AS DIFF2_I1TS
FROM mining.seqfcn
SEQUENCE BY TS;

DIFF2_I1TS
----------------------
                     ?
                     ?
              -.000231
              -.000031
               .000191

--- 5 row(s) selected.
```

EXP Function

This function returns the exponential value (to the base e) of a numeric value expression.

EXP is a Trafodion SQL extension.

```
EXP (numeric-expression)
```

numeric-expression

is an SQL numeric value expression that specifies the value for the argument of the EXP function. See "Numeric Value Expressions" (page 218).

The minimum input value must be between -744.4400719 and -744.4400720.

The maximum input value must be between 709.78271289338404 and 709.78271289338405.

Examples of EXP

- This function returns the value 3.49034295746184128E+000, or approximately 3.4903:

  ```
  EXP (1.25)
  ```

- This function returns the value 2.0. The function EXP is the inverse of the function LOG:

  ```
  EXP (LOG(2.0))
  ```

EXPLAIN Function

- "Considerations for EXPLAIN Function"
- "Examples of EXPLAIN Function"

The EXPLAIN function is a table-valued stored function that generates a result table describing an access plan for a SELECT, INSERT, DELETE, or UPDATE statement. See "Result of the EXPLAIN Function" (page 343).

The EXPLAIN function can be specified as a table reference (*table*) in the FROM clause of a SELECT statement if it is preceded by the keyword TABLE and surrounded by parentheses.

For information on the EXPLAIN statement, see "EXPLAIN Statement" (page 101).

```
EXPLAIN (module,'statement-pattern')

module is:
   'module-name' | NULL
```

'*module-name*'

Reserved for future use.

The module name is enclosed in single quotes and is case-sensitive. If a module name is uppercase, the value you specify within single quotes must be uppercase. For example: 'MYCAT.MYSCH.MYPROG'

NULL

explains statements prepared in the session.

'*statement-pattern*'

A statement pattern is enclosed in single quotes and is case-sensitive. The statement name must be in uppercase, unless you delimit the statement name in a PREPARE statement.

Considerations for EXPLAIN Function

Using a Statement Pattern

Using a statement pattern is analogous to using a LIKE pattern. You can use the LIKE pattern in the following ways:

```
select * from table (explain(NULL,'S%'));
select * from table (explain(NULL,'S1'));
select * from table (explain(NULL,'%1'));
```

However, you cannot use the LIKE pattern in this way:

```
SELECT * FROM TABLE (EXPLAIN (NULL, '%'))
```

This statement returns the EXPLAIN result for all prepared statements whose names begin with the uppercase letter 'S':

```
SELECT * FROM TABLE (EXPLAIN (NULL,'S%'))
```

If the statement pattern does not find any matching statement names, no rows are returned as the result of the SELECT statement.

Obtaining an EXPLAIN Plan While Queries Are Running

Trafodion SQL provides the ability to capture an EXPLAIN plan for a query at any time while the query is running with the QID option. By default, this behavior is disabled for a Trafodion session.

NOTE: Enable this feature before you start preparing and executing queries.

After this feature is enabled, use the following syntax in an EXPLAIN function to get the query execution plan of a running query:

```
SELECT * FROM TABLE (EXPLAIN(NULL, 'QID=qid'))
```

qid is a case-sensitive identifier, which represents the query ID. For example:

```
'QID=MXID0100101119421210365940005336900000085905admin00_2605_S1'
```

The EXPLAIN function or statement returns the plan that was generated when the query was prepared. EXPLAIN for QID retrieves all the information from the original plan of the executing query. The plan is available until the query finishes executing and is removed or deallocated.

Result of the EXPLAIN Function

The result table of the EXPLAIN function describes the access plans for SELECT, INSERT, DELETE, or UPDATE statements.

In this description of the result of the EXPLAIN function, an operator tree is a structure that represents operators used in an access plan as nodes, with at most one parent node for each node in the tree, and with only one root node.

A node of an operator tree is a point in the tree that represents an event (involving an operator) in a plan. Each node might have subordinate nodes—that is, each event might generate a subordinate event or events in the plan.

Column Name	Data Type	Description
MODULE_NAME	CHAR(60)	Reserved for future use.
STATEMENT_ NAME	CHAR(60)	Statement name; truncated on the right if longer than 60 characters.
PLAN_ID	LARGEINT	Unique system-generated plan ID automatically assigned by Trafodion SQL; generated at compile time.
SEQ_NUM	INT	Sequence number of the current operator in the operator tree; indicates the sequence in which the operator tree is generated.
OPERATOR	CHAR(30)	Current operator type.
LEFT_CHILD_ SEQ_NUM	INT	Sequence number for the first child operator of the current operator; null if node has no child operators.
RIGHT_CHILD_ SEQ_NUM	INT	Sequence number for the second child operator of the current operator; null if node does not have a second child.
TNAME	CHAR(60)	For operators in scan group, full name of base table, truncated on the right if too long for column. If correlation name differs from table name, simple correlation name first and then table name in parentheses.
CARDINALITY	REAL	Estimated number of rows that will be returned by the current operator. Cardinality appears as ROWS/REQUEST in some forms of EXPLAIN output. For the right child of a nested join, multiply the cardinality by the number of requests to get the total number of rows produced by this operator.
OPERATOR_COST	REAL	Estimated cost associated with the current operator to execute the operator.
TOTAL_COST	REAL	Estimated cost associated with the current operator to execute the operator, including the cost of all subtrees in the operator tree.
DETAIL_COST	VARCHAR (200)	Cost vector of five items, described in the next table.
DESCRIPTION	VARCHAR (3000)	Additional information about the operator.

The DETAIL_COST column of the EXPLAIN function results contains these cost factors:

CPU_TIME	An estimate of the number of seconds of processor time it might take to execute the instructions for this operator. A value of 1.0 is 1 second.
IO_TIME	An estimate of the number of seconds of I/O time (seeks plus data transfer) to perform the I/O for this operator.
MSG_TIME	An estimate of the number of seconds it takes for the messaging for this operator. The estimate includes the time for the number of local and remote messages and the amount of data sent.
IDLETIME	An estimate of the number of seconds to wait for an event to happen. The estimate includes the amount of time to open a table or start an ESP process.
PROBES	The number of times the operator will be executed. Usually, this value is 1, but it can be greater when you have, for example, an inner scan of a nested-loop join.

Examples of EXPLAIN Function

Display the specified columns in the result table of the EXPLAIN function for the prepared statement REGION:

```
>>select seq_num, operator, operator_cost from table (explain (null, 'REG'));

SEQ_NUM     OPERATOR                       OPERATOR_COST
----------- ------------------------------ ---------------
          1 TRAFODION_SCAN                      0.43691027
          2 ROOT                                       0.0

--- 2 row(s) selected.
>>log;
```

The example displays only part of the result table of the EXPLAIN function. It first uses the EXPLAIN function to generate the table and then selects the desired columns.

EXTRACT Function

The EXTRACT function extracts a datetime field from a datetime or interval value expression. It returns an exact numeric value.

```
EXTRACT (datetime-field FROM extract-source)

datetime-field is:
   YEAR | MONTH | DAY | HOUR | MINUTE | SECOND

extract-source is:
   datetime-expression | interval-expression
```

See "Datetime Value Expressions" (page 212) and "Interval Value Expressions" (page 215).

Examples of EXTRACT

- Extract the year from a DATE value:

  ```
  EXTRACT (YEAR FROM DATE '2007-09-28')
  ```

  The result is 2007.

- Extract the year from an INTERVAL value:

  ```
  EXTRACT (YEAR FROM INTERVAL '01-09' YEAR TO MONTH)
  ```

  The result is 1.

FLOOR Function

The FLOOR function returns the largest integer, represented as a FLOAT data type, less than or equal to a numeric value expression.

FLOOR is a Trafodion SQL extension.

```
FLOOR (numeric-expression)
```

numeric-expression

is an SQL numeric value expression that specifies the value for the argument of the FLOOR function. See "Numeric Value Expressions" (page 218).

Examples of FLOOR

This function returns the integer value 2.00000000000000000E+000, represented as a FLOAT data type:

```
FLOOR (2.25)
```

HOUR Function

The HOUR function converts a TIME or TIMESTAMP expression into an INTEGER value in the range 0 through 23 that represents the corresponding hour of the day.

HOUR is a Trafodion SQL extension.

```
HOUR (datetime-expression)
```

datetime-expression

is an expression that evaluates to a datetime value of type TIME or TIMESTAMP. See "Datetime Value Expressions" (page 212).

Example of HOUR

Return an integer that represents the hour of the day from the SHIP_TIMESTAMP column in the PROJECT table:

```
SELECT start_date, ship_timestamp, HOUR(ship_timestamp)
FROM persnl.project
WHERE projcode = 1000;

Start/Date  Time/Shipped                (EXPR)
----------  --------------------------  ------
2007-04-10  2007-04-21 08:15:00.000000       8
```

INSERT Function

The INSERT function returns a character string where a specified number of characters within the character string has been deleted, beginning at a specified start position, and where another character string has been inserted at the start position. Every character, including multibyte characters, is treated as one character.

INSERT is a Trafodion SQL extension.

```
INSERT (char-expr-1, start, length, char-expr-2)
```

char-expr-1, *char-expr-2*
: are SQL character value expressions (of data type CHAR or VARCHAR) that specify two strings of characters. The character string *char-expr-2* is inserted into the character string *char-expr-1*. See "Character Value Expressions" (page 211).

start
: specifies the starting position *start* within *char-expr-1* at which to start deleting *length* number of characters. After the deletion, the character string *char-expr-2* is inserted into the character string *char-expr-1*, beginning at the start position specified by the number *start*. The number *start* must be a value greater than zero of exact numeric data type and with a scale of zero.

length
: specifies the number of characters to delete from *char-expr-1*. The number *length* must be a value greater than or equal to zero of exact numeric data type and with a scale of zero. *length* must be less than or equal to the length of *char-expr-1*.

Examples of INSERT

Suppose that your JOB table includes an entry for a sales representative. Use the INSERT function to change SALESREP to SALES REP:

```
UPDATE persnl.job
SET jobdesc = INSERT (jobdesc, 6, 3, ' REP')
WHERE jobdesc = 'SALESREP';
```

Now check the row you updated:

```
SELECT jobdesc FROM persnl.job
WHERE jobdesc = 'SALES REP';

Job Description
------------------
SALES REP

--- 1 row(s) selected.
```

ISNULL Function

The ISNULL function returns the value of the first argument if it is not null, otherwise it returns the value of the second argument. Both expressions must be of comparable types.

ISNULL is a Trafodion SQL extension.

```
ISNULL(ck_expr, repl_value)
```

ck_expr

an expression of any valid SQL data type.

repl_value

an expression of any valid SQL data type, but must be a comparable type with that of *ck_expr*.

Examples of ISNULL

- This function returns a 0 instead of a null if `value` is null.

  ```
  ISNULL(value,0)
  ```

- This function returns the date constant if `date_col` is null.

  ```
  ISNULL(date_col, DATE '2006-01-01')
  ```

- This function returns 'Smith' if the string column `last_name` is null.

  ```
  ISNULL(last_name, 'Smith')
  ```

JULIANTIMESTAMP Function

The JULIANTIMESTAMP function converts a datetime value into a 64-bit Julian timestamp value that represents the number of microseconds that have elapsed between 4713 B.C., January 1, 00:00, and the specified datetime value. JULIANTIMESTAMP returns a value of data type LARGEINT.

The function is evaluated once when the query starts execution and is not reevaluated (even if it is a long running query).

JULIANTIMESTAMP is a Trafodion SQL extension.

```
JULIANTIMESTAMP (datetime-expression)
```

datetime-expression

is an expression that evaluates to a value of type DATE, TIME, or TIMESTAMP. If *datetime-expression* does not contain all the fields from YEAR through SECOND, Trafodion SQL extends the value before converting it to a Julian timestamp. Datetime fields to the left of the specified datetime value are set to current date fields. Datetime fields to the right of the specified datetime value are set to zero. See "Datetime Value Expressions" (page 212).

Considerations for JULIANTIMESTAMP

The *datetime-expression* value must be a date or timestamp value from the beginning of year 0001 to the end of year 9999.

Examples of JULIANTIMESTAMP

The PROJECT table consists of five columns using the data types NUMERIC, VARCHAR, DATE, TIMESTAMP, and INTERVAL.

- Convert the TIMESTAMP value into a Julian timestamp representation:

```
SELECT ship_timestamp, JULIANTIMESTAMP (ship_timestamp)
FROM persnl.project
WHERE projcode = 1000;

SHIP_TIMESTAMP              (EXPR)
--------------------------  --------------------
2008-04-21 08:15:00.000000    212075525700000000

--- 1 row(s) selected.
```

- Convert the DATE value into a Julian timestamp representation:

```
SELECT start_date, JULIANTIMESTAMP (start_date)
FROM persnl.project
WHERE projcode = 1000;

START_DATE  (EXPR)
----------  --------------------
2008-04-10    212074545600000000

--- 1 row(s) selected.
```

LASTNOTNULL Function

The LASTNOTNULL function is a sequence function that returns the last nonnull value of a column in an intermediate result table ordered by a SEQUENCE BY clause in a SELECT statement. See "SEQUENCE BY Clause" (page 268).

LASTNOTNULL is a Trafodion SQL extension.

```
LASTNOTNULL (column-expression)
```

column-expression

specifies a derived column determined by the evaluation of the column expression. If only null values have been returned, LASTNOTNULL returns null.

Example of LASTNOTNULL

Return the last nonnull value of a column:

```
SELECT LASTNOTNULL (I1) AS LASTNOTNULL
FROM mining.seqfcn SEQUENCE BY TS;

LASTNOTNULL
-----------
       6215
       6215
      19058
      19058
      11966

--- 5 row(s) selected.
```

LCASE Function

The LCASE function downshifts alphanumeric characters. For non-alphanumeric characters, LCASE returns the same character. LCASE can appear anywhere in a query where a value can be used, such as in a select list, an ON clause, a WHERE clause, a HAVING clause, a LIKE predicate, an expression, or as qualifying a new value in an UPDATE or INSERT statement. The result returned by the LCASE function is equal to the result returned by the "LOWER Function" (page 357).

LCASE returns a string of fixed-length or variable-length character data, depending on the data type of the input string.

LCASE is a Trafodion SQL extension.

```
LCASE (character-expression)
```

character-expression

is an SQL character value expression that specifies a string of characters to downshift. See "Character Value Expressions" (page 211).

Example of LCASE

Suppose that your CUSTOMER table includes an entry for Hotel Oregon. Select the column CUSTNAME and return in uppercase and lowercase letters by using the UCASE and LCASE functions:

```
SELECT custname,UCASE(custname),LCASE(custname)
FROM sales.customer;

(EXPR)               (EXPR)               (EXPR)
-----------------  -------------------  ------------------
...                ...                  ...
Hotel Oregon       HOTEL OREGON         hotel oregon

--- 17 row(s) selected.
```

See "UCASE Function" (page 422).

LEFT Function

The LEFT function returns the leftmost specified number of characters from a character expression. Every character, including multibyte characters, is treated as one character.

LEFT is a Trafodion SQL extension.

```
LEFT (character-expr, count)
```

character-expr

specifies the source string from which to return the leftmost specified number of characters. The source string is an SQL character value expression. The operand is the result of evaluating *character-expr*. See "Character Value Expressions" (page 211).

count

specifies the number of characters to return from *character-expr*. The number *count* must be a value of exact numeric data type greater than or equal to 0 with a scale of zero.

Examples of LEFT

- Return 'Robert':

```
LEFT ('Robert John Smith', 6)
```

- Use the LEFT function to append the company name to the job descriptions:

```
UPDATE persnl.job
SET jobdesc = LEFT (jobdesc, 11) ||' COMNET';

SELECT jobdesc FROM persnl.job;

Job Description
------------------

MANAGER COMNET
PRODUCTION  COMNET
ASSEMBLER COMNET
SALESREP COMNET
SYSTEM ANAL COMNET
ENGINEER COMNET
PROGRAMMER COMNET
ACCOUNTANT COMNET
ADMINISTRAT COMNET
SECRETARY COMNET

--- 10 row(s) selected.
```

LOCATE Function

The LOCATE function searches for a given substring in a character string. If the substring is found, Trafodion SQL returns the character position of the substring within the string. Every character, including multibyte characters, is treated as one character. The result returned by the LOCATE function is equal to the result returned by the "POSITION Function" (page 381).

LOCATE is a Trafodion SQL extension.

```
LOCATE (substring-expression,source-expression)
```

substring-expression

is an SQL character value expression that specifies the substring to search for in *source-expression*. The *substring-expression* cannot be NULL. See "Character Value Expressions" (page 211).

source-expression

is an SQL character value expression that specifies the source string. The *source-expression* cannot be NULL. See "Character Value Expressions" (page 211).

Trafodion SQL returns the result as a 2-byte signed integer with a scale of zero. If *substring-expression* is not found in *source-expression*, Trafodion SQL returns 0.

Considerations for LOCATE

Result of LOCATE

If the length of *source-expression* is zero and the length of *substring-expression* is greater than zero, Trafodion SQL returns 0. If the length of *substring-expression* is zero, Trafodion SQL returns 1.

If the length of *substring-expression* is greater than the length of *source-expression*, Trafodion SQL returns 0. If *source-expression* is a null value, Trafodion SQL returns a null value.

Using UCASE

To ignore case in the search, use the UCASE function (or the LCASE function) for both the *substring-expression* and the *source-expression*.

Examples of LOCATE

- Return the value 8 for the position of the substring `'John'` within the string:

  ```
  LOCATE ('John','Robert John Smith')
  ```

- Suppose that the EMPLOYEE table has an EMPNAME column that contains both the first and last names. This SELECT statement returns all records in table EMPLOYEE that contain the substring 'SMITH', regardless of whether the column value is in uppercase or lowercase characters:

  ```
  SELECT * FROM persnl.employee
    WHERE LOCATE ('SMITH',UCASE(empname)) > 0 ;
  ```

LOG Function

The LOG function returns the natural logarithm of a numeric value expression.

LOG is a Trafodion SQL extension.

```
LOG (numeric-expression)
```

numeric-expression

is an SQL numeric value expression that specifies the value for the argument of the LOG function. The value of the argument must be greater than zero. See "Numeric Value Expressions" (page 218).

Example of LOG

This function returns the value 6.93147180559945344E-001, or approximately 0.69315:

```
LOG (2.0)
```

LOG10 Function

The LOG10 function returns the base 10 logarithm of a numeric value expression.

LOG10 is a Trafodion SQL extension.

```
LOG10 (numeric-expression)
```

numeric-expression

is an SQL numeric value expression that specifies the value for the argument of the LOG10 function. The value of the argument must be greater than zero. See "Numeric Value Expressions" (page 218).

Example of LOG10

This function returns the value 1.39794000867203776E+000, or approximately 1.3979:

```
LOG10 (25)
```

LOWER Function

- "Considerations for LOWER"
- "Example of LOWER"

The LOWER function downshifts alphanumeric characters. For non-alphanumeric characters, LOWER returns the same character. LOWER can appear anywhere in a query where a value can be used, such as in a select list, an ON clause, a WHERE clause, a HAVING clause, a LIKE predicate, an expression, or as qualifying a new value in an UPDATE or INSERT statement. The result returned by the LOWER function is equal to the result returned by the "LCASE Function" (page 352).

LOWER returns a string of fixed-length or variable-length character data, depending on the data type of the input string.

```
LOWER (character-expression)
```

character-expression

is an SQL character value expression that specifies a string of characters to downshift. See "Character Value Expressions" (page 211).

Considerations for LOWER

For a UTF8 character expression, the LOWER function downshifts all the uppercase or title case characters in a given string to lowercase and returns a character string with the same data type and character set as the argument.

A lower case character is a character that has the "alphabetic" property in Unicode Standard 2 whose Unicode name includes lower. An uppercase character is a character that has the "alphabetic" property in the Unicode Standard 2 and whose Unicode name includes *upper*. A title case character is a character that has the Unicode "alphabetic" property and whose Unicode name includes *title*.

Example of LOWER

Suppose that your CUSTOMER table includes an entry for Hotel Oregon. Select the column CUSTNAME and return the result in uppercase and lowercase letters by using the UPPER and LOWER functions:

```
SELECT custname,UPPER(custname),LOWER(custname)
FROM sales.customer;

(EXPR)               (EXPR)               (EXPR)
-----------------  -------------------  ------------------
...                ...                  ...
Hotel Oregon       HOTEL OREGON         hotel oregon

--- 17 row(s) selected.
```

See "UPPER Function" (page 423).

LPAD Function

The LPAD function pads the left side of a string with the specified string. Every character in the string, including multibyte characters, is treated as one character.

LPAD is a Trafodion SQL extension.

```
LPAD (str, len [,padstr])
```

str

can be an expression. See "Character Value Expressions" (page 211).

len

identifies the desired number of characters to be returned and can be an expression but must be an integral value. If *len* is equal to the length of the string, no change is made. If *len* is smaller than the string size, the string is truncated.

pad-character

can be an expression and may be a string.

Examples of LPAD

- This function returns ' kite':

```
lpad('kite', 7)
```

- This function returns 'ki':

```
lpad('kite', 2)
```

- This function returns '0000kite':

```
lpad('kite', 8, '0')
```

- This function returns 'go fly a kite':

```
lpad('go fly a kite', 13, 'z')
```

- This function returns 'John,John, go fly a kite'':

```
lpad('go fly a kite', 23, 'John,')
```

LTRIM Function

The LTRIM function removes leading spaces from a character string. If you must remove any leading character other than space, use the TRIM function and specify the value of the character. See the"TRIM Function" (page 421).

LTRIM is a Trafodion SQL extension.

```
LTRIM (character-expression)
```

character-expression

is an SQL character value expression and specifies the string from which to trim leading spaces. See "Character Value Expressions" (page 211).

Considerations for LTRIM

Result of LTRIM

The result is always of type VARCHAR, with maximum length equal to the fixed length or maximum variable length of *character-expression*.

Example of LTRIM

Return 'Robert_____ ':

```
LTRIM ('    Robert    ')
```

See "TRIM Function" (page 421) and "RTRIM Function" (page 396).

MAX/MAXIMUM Function

MAX is an aggregate function that returns the maximum value within a set of values. MAXIMUM is the equivalent of MAX wherever the function name MAX appears within a statement. The data type of the result is the same as the data type of the argument.

```
MAX | MAXIMUM ([ALL | DISTINCT] expression)
```

ALL | DISTINCT

specifies whether duplicate values are included in the computation of the maximum of the *expression*. The default option is ALL, which causes duplicate values to be included. If you specify DISTINCT, duplicate values are eliminated before the MAX/MAXIMUM function is applied.

expression

specifies an expression that determines the values to include in the computation of the maximum. The *expression* cannot contain an aggregate function or a subquery. The DISTINCT clause specifies that the MAX/MAXIMUM function operates on distinct values from the one-column table derived from the evaluation of *expression*. All nulls are eliminated before the function is applied to the set of values. If the result table is empty, MAX/MAXIMUM returns NULL.

See "Expressions" (page 211).

Considerations for MAX/MAXIMUM

Operands of the Expression

The expression includes columns from the rows of the SELECT result table but cannot include an aggregate function. These expressions are valid:

```
MAX (SALARY)
MAX (SALARY * 1.1)
MAX (PARTCOST * QTY_ORDERED)
```

Example of MAX/MAXIMUM

Display the maximum value in the SALARY column:

```
SELECT MAX (salary)
FROM persnl.employee;

(EXPR)
-----------
  175500.00

--- 1 row(s) selected.
```

MIN Function

MIN is an aggregate function that returns the minimum value within a set of values. The data type of the result is the same as the data type of the argument.

```
MIN ([ALL | DISTINCT] expression)
```

ALL | DISTINCT

specifies whether duplicate values are included in the computation of the minimum of the *expression*. The default option is ALL, which causes duplicate values to be included. If you specify DISTINCT, duplicate values are eliminated before the MIN function is applied.

expression

specifies an expression that determines the values to include in the computation of the minimum. The *expression* cannot contain an aggregate function or a subquery. The DISTINCT clause specifies that the MIN function operates on distinct values from the one-column table derived from the evaluation of *expression*. All nulls are eliminated before the function is applied to the set of values. If the result table is empty, MIN returns NULL.

See "Expressions" (page 211).

Considerations for MIN

Operands of the Expression

The expression includes columns from the rows of the SELECT result table—but cannot include an aggregate function. These expressions are valid:

```
MIN (SALARY)
MIN (SALARY * 1.1)
MIN (PARTCOST * QTY_ORDERED)
```

Example of MIN

Display the minimum value in the SALARY column:

```
SELECT MIN (salary)
FROM persnl.employee;

(EXPR)
-----------
   17000.00

--- 1 row(s) selected.
```

MINUTE Function

The MINUTE function converts a TIME or TIMESTAMP expression into an INTEGER value, in the range 0 through 59, that represents the corresponding minute of the hour.

MINUTE is a Trafodion SQL extension.

```
MINUTE (datetime-expression)
```

datetime-expression

is an expression that evaluates to a datetime value of type TIME or TIMESTAMP. See "Datetime Value Expressions" (page 212).

Example of MINUTE

Return an integer that represents the minute of the hour from the SHIP_TIMESTAMP column in the PROJECT table:

```
SELECT start_date, ship_timestamp, MINUTE(ship_timestamp)
FROM persnl.project
WHERE projcode = 1000;

Start/Date  Time/Shipped                (EXPR)
----------  --------------------------  ------
2008-04-10  2008-04-21 08:15:00.000000      15
```

MOD Function

The MOD function returns the remainder (modulus) of an integer value expression divided by an integer value expression.

MOD is a Trafodion SQL extension.

```
MOD (integer-expression-1,integer-expression-2)
```

integer-expression-1

is an SQL numeric value expression of data type SMALLINT, INTEGER, or LARGEINT that specifies the value for the dividend argument of the MOD function.

integer-expression-2

is an SQL numeric value expression of data type SMALLINT, INTEGER, or LARGEINT that specifies the value for the divisor argument of the MOD function. The divisor argument cannot be zero.

See "Numeric Value Expressions" (page 218).

Example of MOD

This function returns the value 2 as the remainder or modulus:

```
MOD (11,3)
```

MONTH Function

The MONTH function converts a DATE or TIMESTAMP expression into an INTEGER value in the range 1 through 12 that represents the corresponding month of the year.

MONTH is a Trafodion SQL extension.

```
MONTH (datetime-expression)
```

datetime-expression

is an expression that evaluates to a datetime value of type DATE or TIMESTAMP. See "Datetime Value Expressions" (page 212).

Example of MONTH

Return an integer that represents the month of the year from the START_DATE column in the PROJECT table:

```
SELECT start_date, ship_timestamp, MONTH(start_date)
FROM persnl.project
WHERE projcode = 1000;

Start/Date  Time/Shipped                 (EXPR)
----------  --------------------------  ------
2008-04-10  2008-04-21 08:15:00.000000       4
```

MONTHNAME Function

The MONTHNAME function converts a DATE or TIMESTAMP expression into a character literal that is the name of the month of the year (January, February, and so on).

MONTHNAME is a Trafodion SQL extension.

```
MONTHNAME (datetime-expression)
```

datetime-expression

is an expression that evaluates to a datetime value of type DATE or TIMESTAMP. See "Datetime Value Expressions" (page 212).

Considerations for MONTHNAME

The MONTHNAME function returns the name of the month in ISO8859-1.

Example of MONTHNAME

Return a character literal that is the month of the year from the START_DATE column in the PROJECT table:

```
SELECT start_date, ship_timestamp, MONTHNAME(start_date)
FROM persnl.project
WHERE projcode = 1000;

Start/Date  Time/Shipped                 (EXPR)
----------  --------------------------  ---------
2008-04-10  2008-04-21 08:15:00.000000  April
```

MOVINGAVG Function

The MOVINGAVG function is a sequence function that returns the average of nonnull values of a column in the current window of an intermediate result table ordered by a SEQUENCE BY clause in a SELECT statement. See "SEQUENCE BY Clause" (page 268).

MOVINGAVG is a Trafodion SQL extension.

```
MOVINGAVG (column-expression,integer-expression [,max-rows])
```

column-expression

specifies a derived column determined by the evaluation of the column expression.

integer-expression

is an SQL numeric value expression of signed data type SMALLINT or INTEGER that specifies the current window. The current window is defined as the current row and the previous (*integer-expression* - 1) rows.

max-rows

is an SQL numeric value expression of signed data type SMALLINT or INTEGER that specifies the maximum number of rows in the current window.

Note these considerations for the window size:

- The actual value for the window size is the minimum of integer-*expression* and *max-rows*.
- If these conditions are met, MOVINGAVG returns the same result as RUNNINGAVG:
  - The *integer-expression* is out of range, and *max-rows* is not specified. This condition includes the case in which both *integer-expression* and *max-rows* are larger than the result table.
  - The minimum of *integer-expression* and *max-rows* is out of range. In this case, *integer-expression* could be within range, but *max-rows* might be the minimum value of the two and be out of range (for example, a negative number).
- The number of rows is out of range if it is larger than the size of the result table, negative, or NULL.

Example of MOVINGAVG

Return the average of nonnull values of a column in the current window of three rows:

```
create table db.mining.seqfcn (I1 integer, ts timestamp);
SELECT MOVINGAVG (I1,3) AS MOVINGAVG3
FROM mining.seqfcn
SEQUENCE BY TS;

I1              TS
6215            TIMESTAMP '1950-03-05 08:32:09'
28174           TIMESTAMP '1951-02-15 14:35:49'
null            TIMESTAMP '1955-05-18 08:40:10'
4597            TIMESTAMP '1960-09-19 14:40:39'
11966           TIMESTAMP '1964-05-01 16:41:02'

MOVINGAVG3
----------------------
                 6215
                17194
                17194
                16385
                 8281

--- 5 row(s) selected.
```

MOVINGCOUNT Function

The MOVINGCOUNT function is a sequence function that returns the number of nonnull values of a column in the current window of an intermediate result table ordered by a SEQUENCE BY clause in a SELECT statement. See "SEQUENCE BY Clause" (page 268).

MOVINGCOUNT is a Trafodion SQL extension.

```
MOVINGCOUNT (column-expression,integer-expression [,max-rows])
```

column-expression

specifies a derived column determined by the evaluation of the column expression.

integer-expression

is an SQL numeric value expression of signed data type SMALLINT or INTEGER that specifies the current window. The current window is defined as the current row and the previous (*integer-expression* - 1) rows.

max-rows

is an SQL numeric value expression of signed data type SMALLINT or INTEGER that specifies the maximum number of rows in the current window.

Note these considerations for the window size:

- The actual value for the window size is the minimum of *integer-expression* and *max-rows*.
- If these conditions are met, MOVINGCOUNT returns the same result as RUNNINGCOUNT:
  - The *integer-expression* is out of range, and *max-rows* is not specified. This condition includes the case in which both *integer-expression* and *max-rows* are larger than the result table.
  - The minimum of *integer-expression* and *max-rows* is out of range. In this case, *integer-expression* could be within range, but *max-rows* might be the minimum value of the two and be out of range (for example, a negative number).
- The number of rows is out of range if it is larger than the size of the result table, negative, or NULL.

Considerations for MOVINGCOUNT

The MOVINGCOUNT sequence function is defined differently from the COUNT aggregate function. If you specify DISTINCT for the COUNT aggregate function, duplicate values are eliminated before COUNT is applied. You cannot specify DISTINCT for the MOVINGCOUNT sequence function; duplicate values are counted.

Example of MOVINGCOUNT

Return the number of nonnull values of a column in the current window of three rows:

```
SELECT MOVINGCOUNT (I1,3) AS MOVINGCOUNT3
FROM mining.seqfcn
SEQUENCE BY TS;

MOVINGCOUNT3
------------
           1
           2
           2
           2
           2

--- 5 row(s) selected.
```

MOVINGMAX Function

The MOVINGMAX function is a sequence function that returns the maximum of nonnull values of a column in the current window of an intermediate result table ordered by a SEQUENCE BY clause in a SELECT statement. See "SEQUENCE BY Clause" (page 268).

MOVINGMAX is a Trafodion SQL extension.

```
MOVINGMAX (column-expression,integer-expression [,max-rows])
```

column-expression

specifies a derived column determined by the evaluation of the column expression.

integer-expression

is an SQL numeric value expression of signed data type SMALLINT or INTEGER that specifies the current window. The current window is defined as the current row and the previous (*integer-expression* - 1) rows.

max-rows

is an SQL numeric value expression of signed data type SMALLINT or INTEGER that specifies the maximum number of rows in the current window.

Note these considerations for the window size:

- The actual value for the window size is the minimum of *integer-expression* and *max-rows*.
- If these conditions are met, MOVINGMAX returns the same result as RUNNINGMAX:
  - The *integer-expression* is out of range, and *max-rows* is not specified. This condition includes the case in which both *integer-expression* and *max-rows* are larger than the result table.
  - The minimum of *integer-expression* and *max-rows* is out of range. In this case, *integer-expression* could be within range, but *max-rows* might be the minimum value of the two and be out of range (for example, a negative number).
- The number of rows is out of range if it is larger than the size of the result table, negative, or NULL.

Example of MOVINGMAX

Return the maximum of nonnull values of a column in the current window of three rows:

```
SELECT MOVINGMAX (I1,3) AS MOVINGMAX3
FROM mining.seqfcn
SEQUENCE BY TS;

MOVINGMAX3
------------
       6215
      28174
      28174
      28174
      11966

--- 5 row(s) selected.
```

MOVINGMIN Function

The MOVINGMIN function is a sequence function that returns the minimum of nonnull values of a column in the current window of an intermediate result table ordered by a SEQUENCE BY clause in a SELECT statement. See "SEQUENCE BY Clause" (page 268).

MOVINGMIN is a Trafodion SQL extension.

```
MOVINGMIN (column-expression,integer-expression [,max-rows])
```

column-expression

specifies a derived column determined by the evaluation of the column expression.

integer-expression

is an SQL numeric value expression of signed data type SMALLINT or INTEGER that specifies the current window. The current window is defined as the current row and the previous (*integer-expression* - 1) rows.

max-rows

is an SQL numeric value expression of signed data type SMALLINT or INTEGER that specifies the maximum number of rows in the current window.

Note these considerations for the window size:

- The actual value for the window size is the minimum of *integer-expression* and *max-rows*.
- If these conditions are met, MOVINGMIN returns the same result as RUNNINGMIN:
  - The *integer-expression* is out of range, and *max-rows* is not specified. This condition includes the case in which both *integer-expression* and *max-rows* are larger than the result table.
  - The minimum of i*nteger-expression* and *max-rows* is out of range. In this case, *integer-expression* could be within range, but *max-rows* might be the minimum value of the two and be out of range (for example, a negative number).
- The number of rows is out of range if it is larger than the size of the result table, negative, or NULL.

Example of MOVINGMIN

Return the minimum of nonnull values of a column in the current window of three rows:

```
SELECT MOVINGMIN (I1,3) AS MOVINGMIN3
FROM mining.seqfcn
SEQUENCE BY TS;

MOVINGMIN3
------------
        6215
        6215
        6215
        4597
        4597

--- 5 row(s) selected.
```

MOVINGSTDDEV Function

The MOVINGSTDDEV function is a sequence function that returns the standard deviation of nonnull values of a column in the current window of an intermediate result table ordered by a SEQUENCE BY clause in a SELECT statement. See "SEQUENCE BY Clause" (page 268).

MOVINGSTDDEV is a Trafodion SQL extension.

```
MOVINGSTDDEV (column-expression,integer-expression
              [,max-rows])
```

column-expression

specifies a derived column determined by the evaluation of the column expression.

integer-expression

is an SQL numeric value expression of signed data type SMALLINT or INTEGER that specifies the current window. The current window is defined as the current row and the previous (*integer-expression* - 1) rows.

max-rows

is an SQL numeric value expression of signed data type SMALLINT or INTEGER that specifies the maximum number of rows in the current window.

Note these considerations for the window size:

- The actual value for the window size is the minimum of *integer-expression* and *max-rows*.
- If these conditions are met, MOVINGSTDDEV returns the same result as RUNNINGSTDDEV:
  - The *integer-expression* is out of range, and *max-rows* is not specified. This condition includes the case in which both *integer-expression* and *max-rows* are larger than the result table.
  - The minimum of *integer-expression* and *max-rows* is out of range. In this case, *integer-expression* could be within range, but *max-rows* might be the minimum value of the two and be out of range (for example, a negative number).
- The number of rows is out of range if it is larger than the size of the result table, negative, or NULL.

Example of MOVINGSTDDEV

Return the standard deviation of nonnull values of a column in the current window of three rows:

```
SELECT MOVINGSTDDEV (I1,3) AS MOVINGSTDDEV3
FROM mining.seqfcn
SEQUENCE BY TS;

MOVINGSTDDEV3
-------------------------
 0.0000000000000000E+000
 1.5527357808075397 6E+004
 1.4802016653145611 2E+004
 1.5115012482076664 0E+004
 6.0362754244649900 8E+003

--- 5 row(s) selected.
```

You can use the CAST function for display purposes. For example:

```
SELECT CAST(MOVINGSTDDEV (I1,3) AS DEC (18,3))
FROM mining.seqfcn
SEQUENCE BY TS;

(EXPR)
--------------------
```

```
              .000
         15527.357
         14802.016
         15115.012
          6036.275

--- 5 row(s) selected.
```

MOVINGSUM Function

The MOVINGSUM function is a sequence function that returns the sum of nonnull values of a column in the current window of an intermediate result table ordered by a SEQUENCE BY clause in a SELECT statement. See "SEQUENCE BY Clause" (page 268).

MOVINGSUM is a Trafodion SQL extension.

```
MOVINGSUM (column-expression,integer-expression [,max-rows])
```

column-expression

specifies a derived column determined by the evaluation of the column expression.

integer-expression

is an SQL numeric value expression of signed data type SMALLINT or INTEGER that specifies the current window. The current window is defined as the current row and the previous (*integer-expression* - 1) rows.

max-rows

is an SQL numeric value expression of signed data type SMALLINT or INTEGER that specifies the maximum number of rows in the current window.

Note these considerations for the window size:

- The actual value for the window size is the minimum of *integer-expression* and *max-rows*.
- If these conditions are met, MOVINGSUM returns the same result as RUNNINGSUM:
  - The *integer-expression* is out of range, and *max-rows* is not specified. This condition includes the case in which both i*nteger-expression* and *max-rows* are larger than the result table.
  - The minimum of *integer-expression* and *max-rows* is out of range. In this case, *integer-expression* could be within range, but *max-rows* might be the minimum value of the two and be out of range (for example, a negative number).
- The number of rows is out of range if it is larger than the size of the result table, negative, or NULL.

Example of MOVINGSUM

Return the sum of nonnull values of a column in the current window of three rows:

```
SELECT MOVINGSUM (I1,3) AS MOVINGSUM3
FROM mining.seqfcn
SEQUENCE BY TS;

MOVINGSUM3
------------
        6215
       34389
       34389
       32771
       16563

--- 5 row(s) selected.
```

MOVINGVARIANCE Function

The MOVINGVARIANCE function is a sequence function that returns the variance of nonnull values of a column in the current window of an intermediate result table ordered by a SEQUENCE BY clause in a SELECT statement. See "SEQUENCE BY Clause" (page 268).

MOVINGVARIANCE is a Trafodion SQL extension.

```
MOVINGVARIANCE (column-expression,integer-expression
                [,max-rows])
```

column-expression

specifies a derived column determined by the evaluation of the column expression.

integer-expression

is an SQL numeric value expression of signed data type SMALLINT or INTEGER that specifies the current window. The current window is defined as the current row and the previous (*integer-expression* - 1) rows.

max-rows

is an SQL numeric value expression of signed data type SMALLINT or INTEGER that specifies the maximum number of rows in the current window.

Note these considerations for the window size:

- The actual value for the window size is the minimum of *integer-expression* and *max-rows*.
- If these conditions are met, MOVINGVARIANCE returns the same result as RUNNINGVARIANCE:
  - The *integer-expression* is out of range, and *max-rows* is not specified. This condition includes the case in which both *integer-expression* and *max-rows* are larger than the result table.
  - The minimum of *integer-expression* and *max-rows* is out of range. In this case, *integer-expression* could be within range, but *max-rows* might be the minimum value of the two and be out of range (for example, a negative number).
- The number of rows is out of range if it is larger than the size of the result table, negative, or NULL.

Example of MOVINGVARIANCE

Return the variance of nonnull values of a column in the current window of three rows:

```
SELECT MOVINGVARIANCE (I1,3) AS MOVINGVARIANCE3
FROM mining.seqfcn
SEQUENCE BY TS;

MOVINGVARIANCE3
------------------------
 0.00000000000000000E+000
 2.41098840499999960E+008
 2.19099696999999968E+008
 2.28463602333333304E+008
 3.64366210000000016E+007

--- 5 row(s) selected.
```

You can use the CAST function for display purposes. For example:

```
SELECT CAST(MOVINGVARIANCE (I1,3) AS DEC (18,3))
FROM mining.seqfcn
SEQUENCE BY TS;

(EXPR)
```

```
--------------------
                .000
       241098840.500
       219099697.000
       228463602.333
        36436621.000

--- 5 row(s) selected.
```

NULLIF Function

The NULLIF function compares the value of two expressions. Both expressions must be of comparable types. The return value is NULL when the two expressions are equal. Otherwise, the return value is the value of the first expression.

```
NULLIF(expr1, expr2)
```

expr1

an expression to be compared.

expr2

an expression to be compared.

The NULLIF(*expr1*, *expr2*) is equivalent to:

```
CASE  WHEN expr1 = expr2
      THEN  NULL
      ELSE  expr1
END
```

NULLIF returns a NULL if both arguments are equal. The return value is the value of the first argument when the two expressions are not equal.

Example of NULLIF

This function returns a null if the *value* is equal to 7. The return value is the value of the first argument when that value is not 7.

```
NULLIF(value,7)
```

NULLIFZERO Function

The NULLIFZERO function returns the value of the expression if that value is not zero. It returns NULL if the value of the expression is zero.

```
NULLIFZERO (expression)
```

expression

specifies a value expression. It must be a numeric data type.

Examples of NULLIFZERO

- This function returns the value of the column named `salary` for each row where the column's value is not zero. It returns a NULL for each row where the column's value is zero.

  ```
  SELECT NULLIFZERO (salary) from employee_tab;
  ```

- This function returns a value of 1 for each row of the table:

  ```
  SELECT NULLIFZERO(1) from employee_tab;
  ```

- This function returns a value of NULL for each row of the table:

  ```
  SELECT NULLIFZERO(0) from employee_tab;
  ```

NVL Function

The NVL function determines if the selected column has a null value and then returns the new-operand value; otherwise the operand value is returned.

```
NVL (operand, new-operand)
```

operand

specifies a value expression.

new-operand

specifies a value expression. *operand* and *new-operand* must be comparable data types.

If *operand* is a null value, NVL returns *new-operand*.

If *operand* is not a null value, NVL returns *operand*.

The *operand* and *new-operand* can be a column name, subquery, Trafodion SQL string functions, math functions, or constant values.

Examples of NVL

- This function returns a value of z:

```
select nvl(cast(null as char(1)), 'z') from (values(1)) x(a);
(EXPR)
------

"z"
--- 1 row(s) selected.
```

- This function returns a value of 1:

```
select nvl(1, 2) from (values(0)) x(a)
(EXPR)
-------

      1

--- 1 row(s) selected.
```

- This function returns a value of 9999999 for the null value in the column named a1:

```
select nvl(a1, 9999999) from t1;
(EXPR)
-------

    123
     34
9999999

--- 3 row(s) selected.

select * from t1;
A1
-------

    123
     34
      ?

--- 3 row(s) selected.
```

OCTET_LENGTH Function

The OCTET_LENGTH function returns the length of a character string in bytes.

```
OCTET_LENGTH (string-value-expression)
```

string-value-expression

specifies the string value expression for which to return the length in bytes. Trafodion SQL returns the result as a 2-byte signed integer with a scale of zero. If *string-value-expression* is null, Trafodion SQL returns a length of zero. See "Character Value Expressions" (page 211).

Considerations for OCTET_LENGTH

CHAR and VARCHAR Operands

For a column declared as fixed CHAR, Trafodion SQL returns the length of that column as the maximum number of storage bytes. For a VARCHAR column, Trafodion SQL returns the length of the string stored in that column as the actual number of storage bytes.

Similarity to CHAR_LENGTH Function

The OCTET_LENGTH and CHAR_LENGTH functions are similar. The OCTET_LENGTH function returns the number of bytes, rather than the number of characters, in the string. This distinction is important for multibyte implementations. For an example of selecting a double-byte column, see "Example of OCTET_LENGTH" (page 378).

Example of OCTET_LENGTH

If a character string is stored as two bytes for each character, this function returns the value 12. Otherwise, the function returns 6:

```
OCTET_LENGTH ('Robert')
```

OFFSET Function

The OFFSET function is a sequence function that retrieves columns from previous rows of an intermediate result table ordered by a SEQUENCE BY clause in a SELECT statement. See "SEQUENCE BY Clause" (page 268). OFFSET is a Trafodion SQL extension.

```
OFFSET (column-expression,number-rows [,max-rows])
```

column-expression

specifies a derived column determined by the evaluation of the column expression.

number-rows

is an SQL numeric value expression of signed data type SMALLINT or INTEGER that specifies the offset as the number of rows from the current row. If the number of rows exceeds *max-rows*, OFFSET returns OFFSET(*column-expression*,*max-rows*). If the number of rows is out of range and *max-rows* is not specified or is out of range, OFFSET returns null. The number of rows is out of range if it is larger than the size of the result table, negative, or NULL.

max-rows

is an SQL numeric value expression of signed data type SMALLINT or INTEGER that specifies the maximum number of rows of the offset.

Example of OFFSET

Retrieve the I1 column offset by three rows:

```
SELECT OFFSET (I1,3) AS OFFSET3
FROM mining.seqfcn
SEQUENCE BY TS;

OFFSET3
------------
           ?
           ?
           ?
        6215
       28174

--- 5 row(s) selected.
```

The first three rows retrieved display null because the offset from the current row does not fall within the result table.

PI Function

The PI function returns the constant value of pi as a floating-point value.

PI is a Trafodion SQL extension.

```
PI()
```

Example of PI

This constant function returns the value 3.14159260000000000E+000:

```
PI()
```

POSITION Function

The POSITION function searches for a given substring in a character string. If the substring is found, Trafodion SQL returns the character position of the substring within the string. Every character, including multibyte characters, is treated as one character. The result returned by the POSITION function is equal to the result returned by the "LOCATE Function" (page 354).

```
POSITION (substring-expression IN source-expression)
```

substring-expression

is an SQL character value expression that specifies the substring to search for in *source-expression*. The *substring-expression* cannot be NULL. See "Character Value Expressions" (page 211).

source-expression

is an SQL character value expression that specifies the source string. The *source-expression* cannot be NULL. See "Character Value Expressions" (page 211).

Trafodion SQL returns the result as a 2-byte signed integer with a scale of zero. If *substring-expression* is not found in *source-expression*, Trafodion SQL returns zero.

Considerations for POSITION

Result of POSITION

If the length of *source-expression* is zero and the length of *substring-expression* is greater than zero, Trafodion SQL returns 0. If the length of *substring-expression* is zero, Trafodion SQL returns 1.

If the length of *substring-expression* is greater than the length of *source-expression*, Trafodion SQL returns zero. If *source-expression* is a null value, Trafodion SQL returns a null value.

Using the UPSHIFT Function

To ignore case in the search, use the UPSHIFT function (or the LOWER function) for both the *substring-expression* and the *source-expression*.

Examples of POSITION

- This function returns the value 8 for the position of the substring `'John'` within the string:

  ```
  POSITION ('John' IN 'Robert John Smith')
  ```

- Suppose that the EMPLOYEE table has an EMPNAME column that contains both the first and last names. Return all records in table EMPLOYEE that contain the substring 'Smith' regardless of whether the column value is in uppercase or lowercase characters:

  ```
  SELECT * FROM persnl.employee
    WHERE POSITION ('SMITH' IN UPSHIFT(empname)) > 0 ;
  ```

POWER Function

The POWER function returns the value of a numeric value expression raised to the power of an integer value expression. You can also use the exponential operator **.

POWER is a Trafodion SQL extension.

```
POWER (numeric-expression-1,numeric-expression-2)
```

numeric-expression-1, *numeric-expression-2*

are SQL numeric value expressions that specify the values for the base and exponent arguments of the POWER function. See "Numeric Value Expressions" (page 218).

If base *numeric-expression-1* is zero, the exponent *numeric-expression-2* must be greater than zero, and the result is zero. If the exponent is zero, the base cannot be 0, and the result is 1. If the base is negative, the exponent must be a value with an exact numeric data type and a scale of zero.

Examples of POWER

- Return the value 15.625:

  ```
  POWER (2.5,3)
  ```

- Return the value 27. The function POWER raised to the power of 2 is the inverse of the function SQRT:

  ```
  POWER (SQRT(27),2)
  ```

QUARTER Function

The QUARTER function converts a DATE or TIMESTAMP expression into an INTEGER value in the range 1 through 4 that represents the corresponding quarter of the year. Quarter 1 represents January 1 through March 31, and so on.

QUARTER is a Trafodion SQL extension.

```
QUARTER (datetime-expression)
```

datetime-expression

is an expression that evaluates to a datetime value of type DATE or TIMESTAMP. See "Datetime Value Expressions" (page 212).

Example of QUARTER

Return an integer that represents the quarter of the year from the START_DATE column in the PROJECT table:

```
SELECT start_date, ship_timestamp, QUARTER(start_date)
FROM persnl.project
WHERE projcode = 1000;

Start/Date  Time/Shipped                 (EXPR)
----------  --------------------------  ------
2008-04-10  2008-04-21 08:15:00.000000       2
```

RADIANS Function

The RADIANS function converts a numeric value expression (expressed in degrees) to the number of radians.

RADIANS is a Trafodion SQL extension.

```
RADIANS (numeric-expression)
```

numeric-expression

is an SQL numeric value expression that specifies the value for the argument of the RADIANS function. See "Numeric Value Expressions" (page 218).

Examples of RADIANS

- Return the value 7.85398150000000000E-001, or approximately 0.78540 in degrees:

  ```
  RADIANS (45)
  ```

- Return the value 45 in degrees. The function DEGREES is the inverse of the function RADIANS.

  ```
  DEGREES (RADIANS (45))
  ```

RANK/RUNNINGRANK Function

The RANK/RUNNINGRANK function is a sequence function that returns the rank of the given value of an intermediate result table ordered by a SEQUENCE BY clause in a SELECT statement. RANK is an alternative syntax for RANK/RUNNINGRANK.

RANK/RUNNINGRANK is a Trafodion extension.

```
RUNNINGRANK(expression) | RANK(expression)
```

expression

specifies the expression on which to perform the rank.

RANK/RUNNINGRANK returns the rank of the expression within the intermediate result table. The definition of rank is as follows:

```
RANK = 1 for the first value of the intermediate result table.

= the previous value of RANK if the previous value of expression is the
same as the current value of expression.

= RUNNINGCOUNT(*) otherwise.
```

In other words, RANK starts at 1. Values that are equal have the same rank. The value of RANK advances to the relative position of the row in the intermediate result when the value changes.

Considerations for RANK/RUNNINGRANK

Sequence Order Dependency

The RUNNINGRANK function is meaningful only when the given expression is the leading column of the SEQUENCE BY clause. This is because the RUNNINGRANK function assumes that the values of expression are in order and that like values are contiguous. If an ascending order is specified for expression in the SEQUENCE BY clause, then the RUNNINGRANK function assigns a rank of 1 to the lowest value of expression. If a descending order is specified for expression in the SEQUENCE BY clause, then the RUNNINGRANK function assigns a rank of 1 to the highest value of expression.

NULL Values

For the purposes of RUNNINGRANK, NULL values are considered to be equal.

Examples of RANK/RUNNINGRANK

Suppose that SEQFCN has been created as:

```
CREATE TABLE cat.sch.seqfcn
(I1 INTEGER,I2 INTEGER);
```

The table SEQFCN has columns I1 and I2 with data:

I1	I2
1	100
3	200
4	100
2	200
5	300
10	null

I1	I2
6	null
8	200

- Return the rank of I1:

```
SELECT I1, RUNNINGRANK (I1) AS RANK

FROM cat.sch.seqfcn

SEQUENCE BY I1;

I1           RANK
-----------  --------------------
          1                     1
          2                     2
          3                     3
          4                     4
          5                     5
          6                     6
          8                     7
         10                     8

--- 8 row(s) selected.
```

- Return the rank of I1 descending:

```
SELECT I1, RUNNINGRANK (I1) AS RANK
FROM cat.sch.seqfcn
SEQUENCE BY I1 DESC;

I1           RANK

-----------  --------------------

         10                     1
          8                     2
          6                     3
          5                     4
          4                     5
          3                     6
          2                     7
          1                     8

--- 8 row(s) selected.
```

- Return the rank of I2, using the alternative RANK syntax:

```
SELECT I2, RANK (I2) AS RANK
FROM cat.sch.seqfcn
SEQUENCE BY I2;

I2           RANK
-----------  --------------------
        100                     1
        100                     1
        200                     3
        200                     3
        200                     3
        300                     6
          ?                     7
          ?                     7
```

```
--- 8 row(s) selected.
```

Notice that the two NULL values received the same rank.

- Return the rank of I2 descending, using the alternative RANK syntax:

```
SELECT I2, RANK (I2) AS RANK
FROM cat.sch.seqfcn
SEQUENCE BY I2 DESC;

I2           RANK
-----------  --------------------
         ?                     1
         ?                     1
       300                     3
       200                     4
       200                     4
       200                     4
       100                     7
       100                     7

--- 8 row(s) selected.
```

- Return the rank of I2 descending, excluding NULL values:

```
SELECT I2, RANK (I2) AS RANK
FROM cat.sch.seqfcn
WHERE I2 IS NOT NULL
SEQUENCE BY I2 DESC;

I2           RANK
-----------  --------------------
       300                     1
       200                     2
       200                     2
       200                     2
       100                     5
       100                     5
--- 6 row(s) selected.
```

REPEAT Function

The REPEAT function returns a character string composed of the evaluation of a character expression repeated a specified number of times.

REPEAT is a Trafodion SQL extension.

```
REPEAT (character-expr, count)
```

character-expr

specifies the source string from which to return the specified number of repeated strings. The source string is an SQL character value expression. The operand is the result of evaluating *character-expr*. See "Character Value Expressions" (page 211).

count

specifies the number of times the source string *character-expr* is to be repeated. The number *count* must be a value greater than or equal to zero of exact numeric data type and with a scale of zero.

Example of REPEAT

Return this quote from Act 5, Scene 3, of King Lear:

```
REPEAT ('Never,', 5)

Never,Never,Never,Never,Never,
```

REPLACE Function

The REPLACE function returns a character string where all occurrences of a specified character string in the original string are replaced with another character string. All three character value expressions must be comparable types. The return value is the VARCHAR type.

REPLACE is a Trafodion SQL extension.

```
REPLACE (char-expr-1, char-expr-2, char-expr-3)
```

char-expr-1, *char-expr-2*, *char-expr-3*

are SQL character value expressions. The operands are the result of evaluating the character expressions. All occurrences of *char-expr-2* in *char-expr-1* are replaced by *char-expr-3*. See "Character Value Expressions" (page 211).

Example of REPLACE

Use the REPLACE function to change job descriptions so that occurrences of the company name are updated:

```
SELECT jobdesc FROM persnl.job;

Job Description
------------------

MANAGER COMNET
PRODUCTION  COMNET
ASSEMBLER COMNET
SALESREP COMNET
SYSTEM ANAL COMNET
...

--- 10 row(s) selected.

UPDATE persnl.job
SET jobdesc = REPLACE (jobdesc, 'COMNET', 'TDMNET');

Job Description
------------------

MANAGER TDMNET
PRODUCTION  TDMNET
ASSEMBLER TDMNET
SALESREP TDMNET
SYSTEM ANAL TDMNET
...

--- 10 row(s) selected.
```

RIGHT Function

The RIGHT function returns the rightmost specified number of characters from a character expression. Every character, including multibyte characters, is treated as one character.

RIGHT is a Trafodion SQL extension.

```
RIGHT (character-expr, count)
```

character-expr

specifies the source string from which to return the rightmost specified number of characters. The source string is an SQL character value expression. The operand is the result of evaluating *character-expr*. See "Character Value Expressions" (page 211).

count

specifies the number of characters to return from *character-expr*. The number count must be a value of exact numeric data type with a scale of zero.

Examples of RIGHT

- Return 'Smith':

  ```
  RIGHT ('Robert John Smith', 5)
  ```

- Suppose that a six-character company literal has been concatenated as the first six characters to the job descriptions in the JOB table. Use the RIGHT function to remove the company literal from the job descriptions:

  ```
  UPDATE persnl.job
  SET jobdesc = RIGHT (jobdesc, 12);
  ```

ROUND Function

The ROUND function returns the value of *numeric_expr* rounded to *num* places to the right of the decimal point.

ROUND is a Trafodion SQL extension.

```
ROUND(numeric_expr [ , num ] )
```

numeric_expr

is an SQL numeric value expression.

num

specifies the number of places to the right of the decimal point for rounding. If *num* is a negative number, all places to the right of the decimal point and *num* places to the left of the decimal point are zeroed. If *num* is not specified or is 0, then all places to the right of the decimal point are zeroed.

For any exact numeric value, the value *numeric_expr* is rounded away from 0 (for example, to x+1 when x.5 is positive and to x-1 when x.5 is negative). For the inexact numeric values (real, float, and double) the value *numeric_expr* is rounded toward the nearest even number.

Examples of ROUND

- This function returns the value of 123.46.

  ```
  ROUND(123.4567,2)
  ```

- This function returns the value of 123.

  ```
  ROUND(123.4567,0)
  ```

- This function returns the value of 120.

  ```
  ROUND(123.4567,-1)
  ```

- This function returns the value of 0.

  ```
  ROUND(999.0,-4)
  ```

- This function returns the value of 1000.

  ```
  ROUND(999.0.-3)
  ```

- This function returns the value of 2.0E+000.

  ```
  ROUND(1.5E+000,0)
  ```

- This function returns the value of 2.0E+00.

  ```
  ROUND(2.5E+000,0)
  ```

- This function returns the value of 1.0E+00.

  ```
  ROUND(1.4E+000,0)
  ```

ROWS SINCE Function

The ROWS SINCE function is a sequence function that returns the number of rows counted since the specified condition was last true in the intermediate result table ordered by a SEQUENCE BY clause in a SELECT statement. See "SEQUENCE BY Clause" (page 268).

ROWS SINCE is a Trafodion SQL extension.

```
ROWS SINCE [INCLUSIVE] (condition [,max-rows])
```

INCLUSIVE

specifies the current row is to be considered. If you specify INCLUSIVE, the condition is evaluated in the current row. Otherwise, the condition is evaluated beginning with the previous row. If you specify INCLUSIVE and the condition is true in the current row, ROWS SINCE returns 0.

condition

specifies a condition to be considered for each row in the result table. Each column in *condition* must be a column that exists in the result table. If the condition has never been true for the result table, ROWS SINCE returns null.

max-rows

is an SQL numeric value expression of signed data type SMALLINT or INTEGER that specifies the maximum number of rows from the current row to consider. If the condition has never been true for *max-rows* from the current row, or if *max-rows* is negative or null, ROWS SINCE returns null.

Considerations for ROWS SINCE

Counting the Rows

If you specify INCLUSIVE, the condition in each row of the result table is evaluated starting with the current row as row 0 (zero) (up to the maximum number of rows or the size of the result table). Otherwise, the condition is evaluated starting with the previous row as row 1.

If a row is reached where the condition is true, ROWS SINCE returns the number of rows counted so far. Otherwise, if the condition is never true within the result table being considered, ROWS SINCE returns null. Trafodion SQL then goes to the next row as the new current row.

Examples of ROWS SINCE

- Return the number of rows since the condition `I1 IS NULL` became true:

```
SELECT ROWS SINCE (I1 IS NULL) AS ROWS_SINCE_NULL
FROM mining.seqfcn
SEQUENCE BY TS;

ROWS_SINCE_NULL
---------------
              ?
              ?
              1
              2
              1

--- 5 row(s) selected.
```

- Return the number of rows since the condition `I1 < I2` became true:

```
SELECT ROWS SINCE (I1<I2), ROWS SINCE INCLUSIVE (I1<I2)
FROM mining.seqfcn
SEQUENCE BY TS;

(EXPR)           (EXPR)
```

```
---------------  ---------------
               ?                0
               1                1
               2                0
               1                1
               2                0

--- 5 row(s) selected.
```

ROWS SINCE CHANGED Function

The ROWS SINCE CHANGED function is a sequence function that returns the number of rows counted since the specified set of values last changed in the intermediate result table ordered by a SEQUENCE BY clause in a SELECT statement. See "SEQUENCE BY Clause" (page 268).

ROWS SINCE CHANGED is a Trafodion SQL extension.

```
ROWS SINCE CHANGED (column-expression-list)
```

column-expression-list

is a comma-separated list that specifies a derived column list determined by the evaluation of the column expression list. ROWS SINCE CHANGED returns the number of rows counted since the values of *column-expression-list* changed.

Considerations for ROWS SINCE CHANGED

Counting the Rows

For the first row in the intermediate result table, the count is 1. For subsequent rows that have the same value for *column-expression-list* as the previous row, the count is 1 plus the count in the previous row. For subsequent rows that have a different value of *column-expression-list* than the previous row, the count is 1.

Examples of ROWS SINCE CHANGED

- Return the number of rows since the value `I1` last changed:

```
SELECT ROWS SINCE CHANGED (I1)
FROM mining.seqfcn
SEQUENCE BY TS;
```

- Return the number of rows since the values `I1` and `ts` last changed:

```
SELECT ROWS SINCE CHANGED (I1, TS)
FROM mining.seqfcn
SEQUENCE BY TS;
```

RPAD Function

The RPAD function pads the right side of a string with the specified string. Every character in the string, including multibyte characters, is treated as one character.

RPAD is a Trafodion SQL extension.

```
RPAD (str, len [,padstr])
```

str
: can be an expression. See "Character Value Expressions" (page 211).

len
: identifies the desired number of characters to be returned and can be an expression but must be an integral value. If *len* is equal to the length of the string, no change is made. If *len* is smaller than the string size, the string is truncated.

pad-character
: can be an expression and may be a string.

Examples of RPAD Function

- This function returns `'kite   '`:

  ```
  rpad('kite', 7)
  ```

- This function returns `'ki'`:

  ```
  rpad('kite', 2)
  ```

- This function returns `'kite0000'`:

  ```
  rpad('kite', 8, '0')
  ```

- This function returns `'go fly a kite'`:

  ```
  rpad('go fly a kite', 13, 'z')
  ```

- This function returns 'go fly a kitez'

  ```
  rpad('go fly a kite', 14, 'z')
  ```

- This function returns `'kitegoflygoflygof'`:

  ```
  rpad('kite', 17, 'gofly' )
  ```

RTRIM Function

The RTRIM function removes trailing spaces from a character string. If you must remove any leading character other than space, use the TRIM function and specify the value of the character. See the "TRIM Function" (page 421).

RTRIM is a Trafodion SQL extension.

```
RTRIM (character-expression)
```

character-expression

is an SQL character value expression and specifies the string from which to trim trailing spaces. See "Character Value Expressions" (page 211).

Considerations for RTRIM

Result of RTRIM

The result is always of type VARCHAR, with maximum length equal to the fixed length or maximum variable length of *character-expression*.

Example of RTRIM

Return ' Robert':

```
RTRIM ('   Robert   ')
```

See "TRIM Function" (page 421) and "LTRIM Function" (page 359).

RUNNINGAVG Function

The RUNNINGAVG function is a sequence function that returns the average of nonnull values of a column up to and including the current row of an intermediate result table ordered by a SEQUENCE BY clause in a SELECT statement. See "SEQUENCE BY Clause" (page 268).

RUNNINGAVG is a Trafodion SQL extension.

```
RUNNINGAVG (column-expression)
```

column-expression

specifies a derived column determined by the evaluation of the column expression. RUNNINGAVG returns the average of nonnull values of *column-expression* up to and including the current row.

Considerations for RUNNINGAVG

Equivalent Result

The result of RUNNINGAVG is equivalent to:

```
RUNNINGSUM(column-expr) / RUNNINGCOUNT(*)
```

Example of RUNNINGAVG

Return the average of nonnull values of I1 up to and including the current row:

```
SELECT RUNNINGAVG (I1) AS AVG_I1
FROM mining.seqfcn
SEQUENCE BY TS;

AVG_I1
--------------------
                6215
               17194
               11463
                9746
               10190

--- 5 row(s) selected.
```

RUNNINGCOUNT Function

The RUNNINGCOUNT function is a sequence function that returns the number of rows up to and including the current row of an intermediate result table ordered by a SEQUENCE BY clause in a SELECT statement. See "SEQUENCE BY Clause" (page 268).

RUNNINGCOUNT is a Trafodion SQL extension.

```
RUNNINGCOUNT {(*) | (column-expression)}
```

*

as an argument causes RUNNINGCOUNT(*) to return the number of rows in the intermediate result table up to and including the current row.

column-expression

specifies a derived column determined by the evaluation of the column expression. If *column-expression* is the argument, RUNNINGCOUNT returns the number of rows containing nonnull values of *column-expression* in the intermediate result table up to and including the current row.

Considerations for RUNNINGCOUNT

No DISTINCT Clause

The RUNNINGCOUNT sequence function is defined differently from the COUNT aggregate function. If you specify DISTINCT for the COUNT aggregate function, duplicate values are eliminated before COUNT is applied. You cannot specify DISTINCT for the RUNNINGCOUNT sequence function; duplicate values are counted.

Example of RUNNINGCOUNT

Return the number of rows that include nonnull values of I1 up to and including the current row:

```
SELECT RUNNINGCOUNT (I1) AS COUNT_I1
FROM mining.seqfcn
SEQUENCE BY TS;

COUNT_I1
------------
           1
           2
           2
           3
           4

--- 5 row(s) selected.
```

RUNNINGMAX Function

The RUNNINGMAX function is a sequence function that returns the maximum of values of a column up to and including the current row of an intermediate result table ordered by a SEQUENCE BY clause in a SELECT statement. See "SEQUENCE BY Clause" (page 268).

RUNNINGMAX is a Trafodion SQL extension.

```
RUNNINGMAX (column-expression)
```

column-expression

specifies a derived column determined by the evaluation of the column expression. RUNNINGMAX returns the maximum of values of *column-expression* up to and including the current row.

Example of RUNNINGMAX

Return the maximum of values of I1 up to and including the current row:

```
SELECT RUNNINGMAX (I1) AS MAX_I1
FROM mining.seqfcn
SEQUENCE BY TS;

MAX_I1
------------
        6215
       28174
       28174
       28174
       28174

--- 5 row(s) selected.
```

RUNNINGMIN Function

The RUNNINGMIN function is a sequence function that returns the minimum of values of a column up to and including the current row of an intermediate result table ordered by a SEQUENCE BY clause in a SELECT statement. See "SEQUENCE BY Clause" (page 268).

RUNNINGMIN is a Trafodion SQL extension.

```
RUNNINGMIN (column-expression)
```

column-expression

specifies a derived column determined by the evaluation of the column expression. RUNNINGMIN returns the minimum of values of *column-expression* up to and including the current row.

Example of RUNNINGMIN

Return the minimum of values of I1 up to and including the current row:

```
SELECT RUNNINGMIN (I1) AS MIN_I1
FROM mining.seqfcn
SEQUENCE BY TS;

MIN_I1
------------
        6215
        6215
        6215
        4597
        4597

--- 5 row(s) selected.
```

RUNNINGRANK Function

See the "RANK/RUNNINGRANK Function" (page 385).

RUNNINGSTDDEV Function

The RUNNINGSTDDEV function is a sequence function that returns the standard deviation of nonnull values of a column up to and including the current row of an intermediate result table ordered by a SEQUENCE BY clause in a SELECT statement. See "SEQUENCE BY Clause" (page 268).

RUNNINGSTDDEV is a Trafodion SQL extension.

```
RUNNINGSTDDEV (column-expression)
```

column-expression

specifies a derived column determined by the evaluation of the column expression. RUNNINGSTDDEV returns the standard deviation of nonnull values of *column-expression* up to and including the current row.

Considerations for RUNNINGSTDDEV

Equivalent Result

The result of RUNNINGSTDDEV is equivalent to:

```
SQRT(RUNNINGVARIANCE(column-expression))
```

Examples of RUNNINGSTDDEV

Return the standard deviation of nonnull values of I1 up to and including the current row:

```
SELECT RUNNINGSTDDEV (I1) AS STDDEV_I1
FROM mining.seqfcn
SEQUENCE BY TS;

STDDEV_I1
-------------------------
 0.00000000000000000E+000
 1.55273578080753976E+004
 1.48020166531456112E+004
 1.25639147428923072E+004
 1.09258501408357232E+004

--- 5 row(s) selected.
```

You can use the CAST function for display purposes. For example:

```
SELECT CAST(RUNNINGSTDDEV (I1) AS DEC (18,3))
FROM mining.seqfcn
SEQUENCE BY TS;

(EXPR)
--------------------
                .000
           15527.357
           14802.016
           12563.914
           10925.850

--- 5 row(s) selected.
```

RUNNINGSUM Function

The RUNNINGSUM function is a sequence function that returns the sum of nonnull values of a column up to and including the current row of an intermediate result table ordered by a SEQUENCE BY clause in a SELECT statement. See "SEQUENCE BY Clause" (page 268).

RUNNINGSUM is a Trafodion SQL extension.

```
RUNNINGSUM (column-expression)
```

column-expression

specifies a derived column determined by the evaluation of the column expression. RUNNINGSUM returns the sum of nonnull values of *column-expression* up to and including the current row.

Example of RUNNINGSUM

Return the sum of nonnull values of I1 up to and including the current row:

```
SELECT RUNNINGSUM (I1) AS SUM_I1
FROM mining.seqfcn
SEQUENCE BY TS;

SUM_I1
--------------------
                6215
               34389
               34389
               38986
               50952

--- 5 row(s) selected.
```

RUNNINGVARIANCE Function

The RUNNINGVARIANCE function is a sequence function that returns the variance of nonnull values of a column up to and including the current row of an intermediate result table ordered by a SEQUENCE BY clause in a SELECT statement. See "SEQUENCE BY Clause" (page 268).

RUNNINGVARIANCE is a Trafodion SQL extension.

```
RUNNINGVARIANCE (column-expression)
```

column-expression

specifies a derived column determined by the evaluation of the column expression. RUNNINGVARIANCE returns the variance of nonnull values of *column-expression* up to and including the current row.

Examples of RUNNINGVARIANCE

Return the variance of nonnull values of I1 up to and including the current row:

```
SELECT RUNNINGVARIANCE (I1) AS VARIANCE_I1
FROM mining.seqfcn
SEQUENCE BY TS;

VARIANCE_I1
-------------------------
 0.00000000000000000E+000
 2.41098840499999960E+008
 2.19099696999999968E+008
 1.57851953666666640E+008
 1.19374201299999980E+008

--- 5 row(s) selected.
```

You can use the CAST function for display purposes. For example:

```
SELECT CAST(RUNNINGVARIANCE (I1) AS DEC (18,3))
FROM mining.seqfcn
SEQUENCE BY TS;

(EXPR)
--------------------
                .000
       241098840.500
       219099697.000
       157851953.666
       119374201.299

--- 5 row(s) selected.
```

SECOND Function

The SECOND function converts a TIME or TIMESTAMP expression into an INTEGER value in the range 0 through 59 that represents the corresponding second of the hour.

SECOND is a Trafodion SQL extension.

```
SECOND (datetime-expression)
```

datetime-expression

is an expression that evaluates to a datetime value of type TIME or TIMESTAMP. See "Datetime Value Expressions" (page 212).

Example of SECOND

Return a NUMERIC value that represents the second of the hour from the SHIP_TIMESTAMP column :

```
SELECT start_date, ship_timestamp, SECOND(ship_timestamp)
FROM persnl.project
WHERE projcode = 1000;

Start/Date  Time/Shipped                (EXPR)
----------  --------------------------  -----------
2008-04-10  2008-04-21 08:15:00.000000      .000000
```

SIGN Function

The SIGN function returns an indicator of the sign of a numeric value expression. If the value is less than zero, the function returns -1 as the indicator. If the value is zero, the function returns 0. If the value is greater than zero, the function returns 1.

SIGN is a Trafodion SQL extension.

```
SIGN (numeric-expression)
```

numeric-expression

is an SQL numeric value expression that specifies the value for the argument of the SIGN function. See "Numeric Value Expressions" (page 218).

Examples of SIGN

- Return the value –1:

  ```
  SIGN (-20 + 12)
  ```

- Return the value 0:

  ```
  SIGN (-20 + 20)
  ```

- Return the value 1:

  ```
  SIGN (-20 + 22)
  ```

SIN Function

The SIN function returns the sine of a numeric value expression, where the expression is an angle expressed in radians.

SIN is a Trafodion SQL extension.

```
SIN (numeric-expression)
```

numeric-expression

is an SQL numeric value expression that specifies the value for the argument of the SIN function. See "Numeric Value Expressions" (page 218).

Example of SIN

This function returns the value 3.42052233254419840E-001, or approximately 0.3420, the sine of 0.3491 (which is 20 degrees):

```
SIN (0.3491)
```

SINH Function

The SINH function returns the hyperbolic sine of a numeric value expression, where the expression is an angle expressed in radians.

SINH is a Trafodion SQL extension.

```
SINH (numeric-expression)
```

numeric-expression

is an SQL numeric value expression that specifies the value for the argument of the SINH function. See "Numeric Value Expressions" (page 218).

Example of SINH

This function returns the value 1.60191908030082560E+000, or approximately 1.6019, the hyperbolic sine of 1.25:

```
SINH (1.25)
```

SPACE Function

The SPACE function returns a character string consisting of a specified number of spaces, each of which is 0x20 or 0x0020, depending on the chosen character set.

SPACE is a Trafodion SQL extension.

```
SPACE (length [,char-set-name])
```

length

specifies the number of characters to be returned. The number *count* must be a value greater than or equal to zero of exact numeric data type and with a scale of zero. *length* cannot exceed 32768 for the ISO8859-1 or UTF8 character sets.

char-set-name

can be ISO88591 or UTF8. If you do not specify this second argument, the default is the default character set.

The returned character string will be of data type VARCHAR associated with the character set specified by *char-set-name*.

Example of SPACE

Return three spaces:

```
SPACE (3)
```

SQRT Function

The SQRT function returns the square root of a numeric value expression.

SQRT is a Trafodion SQL extension.

```
SQRT (numeric-expression)
```

numeric-expression

is an SQL numeric value expression that specifies the value for the argument of the SQRT function. The value of the argument must not be a negative number. See "Numeric Value Expressions" (page 218).

Example of SQRT

This function returns the value 5.19615242270663232E+000, or approximately 5.196:

```
SQRT (27)
```

STDDEV Function

- "Considerations for STDDEV"
- "Examples of STDDEV"

STDDEV is an aggregate function that returns the standard deviation of a set of numbers.

STDDEV is a Trafodion SQL extension.

```
STDDEV ([ALL | DISTINCT] expression [,weight])
```

ALL | DISTINCT

specifies whether duplicate values are included in the computation of the STDDEV of the *expression*. The default option is ALL, which causes duplicate values to be included. If you specify DISTINCT, duplicate values are eliminated before the STDDEV function is applied. If DISTINCT is specified, you cannot specify *weight*.

expression

specifies a numeric value expression that determines the values for which to compute the standard deviation. The *expression* cannot contain an aggregate function or a subquery. The DISTINCT clause specifies that the STDDEV function operates on distinct values from the one-column table derived from the evaluation of *expression*.

weight

specifies a numeric value expression that determines the weights of the values for which to compute the standard deviation. *weight* cannot contain an aggregate function or a subquery. *weight* is defined on the same table as *expression*. The one-column table derived from the evaluation of *expression* and the one-column table derived from the evaluation of *weight* must have the same cardinality.

Considerations for STDDEV

Definition of STDDEV

The standard deviation of a value expression is defined to be the square root of the variance of the expression. See "VARIANCE Function" (page 426).

Because the definition of variance has *N-1* in the denominator of the expression (if weight is not specified), Trafodion SQL returns a system-defined default setting of zero (and no error) if the number of rows in the table, or a group of the table, is equal to 1.

Data Type of the Result

The data type of the result is always DOUBLE PRECISION.

Operands of the Expression

The expression includes columns from the rows of the SELECT result table but cannot include an aggregate function. These are valid:

```
STDDEV (SALARY)
STDDEV (SALARY * 1.1)
STDDEV (PARTCOST * QTY_ORDERED)
```

Nulls

STDDEV is evaluated after eliminating all nulls from the set. If the result table is empty, STDDEV returns NULL.

FLOAT(54) and DOUBLE PRECISION Data

Avoid using large FLOAT(54) or DOUBLE PRECISION values as arguments to STDDEV. If SUM(x * x) exceeds the value of 1.15792089237316192e77 during the computation of STDDEV(x), a numeric overflow occurs.

Examples of STDDEV

- Compute the standard deviation of the salary of the current employees:

```
SELECT STDDEV(salary) AS StdDev_Salary
FROM persnl.employee;

STDDEV_SALARY
-------------------------
 3.57174062500000000E+004

--- 1 row(s) selected.
```

- Compute the standard deviation of the cost of parts in the current inventory:

```
SELECT STDDEV (price * qty_available)
FROM sales.parts;

(EXPR)
-------------------------
 7.13899499999999808E+006

--- 1 row(s) selected.
```

SUBSTRING/SUBSTR Function

The SUBSTRING function extracts a substring out of a given character expression. It returns a character string of data type VARCHAR, with a maximum length equal to the smaller of these two:

- The fixed length of the input string (for CHAR-type strings) or the maximum variable length (for VARCHAR-type strings)
- The value of the length argument (when a constant is specified) or 32708 (when a non-constant is specified)

SUBSTR is equivalent to SUBSTRING.

```
SUBSTRING (character-expr FROM start-position [FOR length])
```

or:

```
SUBSTRING (character-expr,start-position[,length])
```

character-expr

specifies the source string from which to extract the substring. The source string is an SQL character value expression. The operand is the result of evaluating *character-expr*. See "Character Value Expressions" (page 211).

start-position

specifies the starting position *start-position* within *character-expr* at which to start extracting the substring. *start-position* must be a value with an exact numeric data type and a scale of zero.

length

specifies the number of characters to extract from *character-expr*. Keep in mind that every character, including multibyte characters, counts as one character. *length* is the length of the extracted substring and must be a value greater than or equal to zero of exact numeric data type and with a scale of zero. The *length* field is optional, so if you do not specify the substring *length*, all characters starting at *start-position* and continuing until the end of the character expression are returned.

The length field is optional. If you do not specify it, all characters starting at *start-position* and continuing until the end of the *character-expr* are returned.

Alternative Forms

- The SUBSTRING function treats SUBSTRING(*string* FOR *int*) equivalent to SUBSTRING(*string* FROM 1 FOR *int*). The Trafodion database software already supports the ANSI standard form as:

  ```
  SUBSTRING(string FROM int  [ FOR int ])
  ```

- The SUBSTRING function treats SUBSTRING (*string*, Fromint) equivalent to SUBSTRING(*string* FROM *Fromint*). The Trafodion database software already supports SUBSTRING (*string*, *Fromint*, *Forint*) as equivalent to the ANSI standard form:

  ```
  SUBSTRING(string FROM Fromint FOR Forint)
  ```

Considerations for SUBSTRING/SUBSTR

Requirements for the Expression, Length, and Start Position

- The data types of the substring length and the start position must be numeric with a scale of zero. Otherwise, an error is returned.
- If the sum of the start position and the substring length is greater than the length of the character expression, the substring from the start position to the end of the string is returned.

- If the start position is greater than the length of the character expression, an empty string ('') is returned.
- The resulting substring is always of type VARCHAR. If the source character string is an upshifted CHAR or VARCHAR string, the result is an upshifted VARCHAR type.

Examples of SUBSTRING/SUBSTR

- Extract `'Ro'`:

```
SUBSTRING('Robert John Smith' FROM 0 FOR 3)
SUBSTR('Robert John Smith' FROM 0 FOR 3)
```

- Extract 'John':

```
SUBSTRING ('Robert John Smith' FROM 8 FOR 4)
SUBSTR ('Robert John Smith' FROM 8 FOR 4)
```

- Extract 'John Smith':

```
SUBSTRING ('Robert John Smith' FROM 8)
SUBSTR ('Robert John Smith' FROM 8)
```

- Extract 'Robert John Smith':

```
SUBSTRING ('Robert John Smith' FROM 1 FOR 17)
SUBSTR ('Robert John Smith' FROM 1 FOR 17)
```

- Extract 'John Smith':

```
SUBSTRING ('Robert John Smith' FROM 8 FOR 15)
SUBSTR ('Robert John Smith' FROM 8 FOR 15)
```

- Extract 'Ro':

```
SUBSTRING ('Robert John Smith' FROM -2 FOR 5)
SUBSTR ('Robert John Smith' FROM -2 FOR 5)
```

- Extract an empty string '':

```
SUBSTRING ('Robert John Smith' FROM 8 FOR 0)
SUBSTR ('Robert John Smith' FROM 8 FOR 0)
```

SUM Function

SUM is an aggregate function that returns the sum of a set of numbers.

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

ALL | DISTINCT

specifies whether duplicate values are included in the computation of the SUM of the *expression*. The default option is ALL, which causes duplicate values to be included. If you specify DISTINCT, duplicate values are eliminated before the SUM function is applied.

expression

specifies a numeric or interval value expression that determines the values to sum. The *expression* cannot contain an aggregate function or a subquery. The DISTINCT clause specifies that the SUM function operates on distinct values from the one-column table derived from the evaluation of *expression*. All nulls are eliminated before the function is applied to the set of values. If the result table is empty, SUM returns NULL.

See "Expressions" (page 211).

Considerations for SUM

Data Type and Scale of the Result

The data type of the result depends on the data type of the argument. If the argument is an exact numeric type, the result is LARGEINT. If the argument is an approximate numeric type, the result is DOUBLE PRECISION. If the argument is INTERVAL data type, the result is INTERVAL with the same precision as the argument. The scale of the result is the same as the scale of the argument. If the argument has no scale, the result is truncated.

Operands of the Expression

The expression includes columns from the rows of the SELECT result table—but cannot include an aggregate function. The valid expressions are:

```
SUM (SALARY)
SUM (SALARY * 1.1)
SUM (PARTCOST * QTY_ORDERED)
```

Example of SUM

Compute the total value of parts in the current inventory:

```
SELECT SUM (price * qty_available)
FROM sales.parts;

(EXPR)
----------------------
         117683505.96

--- 1 row(s) selected.
```

TAN Function

The TAN function returns the tangent of a numeric value expression, where the expression is an angle expressed in radians.

TAN is a Trafodion SQL extension.

```
TAN (numeric-expression)
```

numeric-expression

is an SQL numeric value expression that specifies the value for the argument of the TAN function. See "Numeric Value Expressions" (page 218).

Example of TAN

This function returns the value 3.64008908293626880E-001, or approximately 0.3640, the tangent of 0.3491 (which is 20 degrees):

```
TAN (0.3491)
```

TANH Function

The TANH function returns the hyperbolic tangent of a numeric value expression, where the expression is an angle expressed in radians.

TANH is a Trafodion SQL extension.

```
TANH (numeric-expression)
```

numeric-expression

is an SQL numeric value expression that specifies the value for the argument of the TANH function. See "Numeric Value Expressions" (page 218).

Example of TANH

- This function returns the value 8.48283639957512960E-001 or approximately 0.8483, the hyperbolic tangent of 1.25:

  ```
  TANH (1.25)
  ```

THIS Function

The THIS function is a sequence function that is used in the ROWS SINCE function to distinguish between the value of the column in the current row and the value of the column in previous rows (in an intermediate result table ordered by a SEQUENCE BY clause in a SELECT statement). See "ROWS SINCE Function" (page 392).

THIS is a Trafodion SQL extension.

```
THIS (column-expression)
```

column-expression

specifies a derived column determined by the evaluation of the column expression. If the value of the expression is null, THIS returns null.

Considerations for THIS

Counting the Rows

You can use the THIS function only within the ROWS SINCE function. For each row, the ROWS SINCE condition is evaluated in two steps:

1. The expression for THIS is evaluated for the current row. This value becomes a constant.
2. The condition is evaluated for the result table, using a combination of the THIS constant and the data for each row in the result table, starting with the previous row as row 1 (up to the maximum number of rows or the size of the result table).

If a row is reached where the condition is true, ROWS SINCE returns the number of rows counted so far. Otherwise, if the condition is never true within the result table being considered, ROWS SINCE returns null. Trafodion SQL then goes to the next row as the new current row and the THIS constant is reevaluated.

Example of THIS

Return the number of rows since the condition I1 less than a previous row became true:

```
SELECT ROWS SINCE (THIS(I1) < I1) AS ROWS_SINCE_THIS
FROM mining.seqfcn
SEQUENCE BY TS;

ROWS_SINCE_THIS
---------------
              ?
              ?
              1
              1
              ?

--- 5 row(s) selected.
```

TIMESTAMPADD Function

The TIMESTAMPADD function adds the interval of time specified by *interval-ind* and *num_expr* to *datetime_expr*. If the specified interval is in years, months, or quarters and the resulting date is not a valid date, the day will be rounded down to the last day of the result month. The type of the *datetime_expr* is returned except when the *interval-ind* contains any time component, in which case a TIMESTAMP is returned.

TIMESTAMPADD is a Trafodion SQL extension.

```
TIMESTAMPADD (interval-ind, num_expr, datetime_expr)
```

interval-ind

is SQL_TSI_YEAR, SQL_TSI_MONTH, SQL_TSI_DAY, SQL_TSI_HOUR, SQL_TSI_MINUTE, SQL_TSI_SECOND, SQL_TSI_QUARTER, or SQL_TSI_WEEK

num_expr

is an SQL exact numeric value expression that specifies how many *interval-ind* units of time are to be added to *datetime_expr*. If *num_expr* has a fractional portion, it is ignored. If *num_expr* is negative, the return value precedes *datetime_expr* by the specified amount of time.

datetime_expr

is an expression that evaluates to a datetime value of type DATE or TIMESTAMP. The type of the *datetime_expr* is returned except when the *interval-ind* contains any time component, in which case a TIMESTAMP is returned.

Examples of TIMESTAMPADD

- This function adds seven days to the date specified in *start-date*:

```
TIMESTAMPADD (SQL_TSI_DAY, 7, start-date)
```

- This function returns the value `DATE '2008-03-06'`:

```
TIMESTAMPADD (SQL_TSI_WEEK, 1, DATE '2008-02-28')
```

- This function returns the value `DATE '1999-02-28'`:

```
TIMESTAMPADD (SQL_TSI_YEAR, -1, DATE '2000-02-29')
```

- This function returns the value `TIMESTAMP '2003-02-28 13:27:35'`:

```
TIMESTAMPADD (SQL_TSI_MONTH, -12, TIMESTAMP '2004-02-29 13:27:35')
```

- This function returns the value `TIMESTAMP '2004-02-28 13:27:35'`:

```
TIMESTAMPADD (SQL_TSI_MONTH, 12, TIMESTAMP '2003-02-28 13:27:35')
```

- This function returns the value `DATE '2008-06-30'`:

```
TIMESTAMPADD (SQL_TSI_QUARTER, -2, DATE '2008-12-31')
```

- This function returns the value `TIMESTAMP '2008-06-30 23:59:55'`:

```
TIMESTAMPADD (SQL_TSI_SECOND, -5, DATE '2008-07-01')
```

TIMESTAMPDIFF Function

The TIMESTAMPDIFF function returns the integer value for the number of *interval-ind* units of time between *startdate* and *enddate*. If *enddate* precedes *startdate*, the return value is negative or zero.

```
TIMESTAMPDIFF (interval-ind, startdate, enddate)
```

interval-ind

is SQL_TSI_YEAR, SQL_TSI_MONTH, SQL_TSI_DAY, SQL_TSI_HOUR, SQL_TSI_MINUTE, SQL_TSI_SECOND, SQL_TSI_QUARTER, or SQL_TSI_WEEK

startdate and *enddate*

are each of type DATE or TIMESTAMP

The method of counting crossed boundaries such as days, minutes, and seconds makes the result given by TIMESTAMPDIFF consistent across all data types. The TIMESTAMPDIFF function makes these boundary assumptions:

- A year begins at the start of January 1.
- A new quarter begins on January 1, April 1, July 1, and October 1.
- A week begins at the start of Sunday.
- A day begins at midnight.

The result is a signed integer value equal to the number of *interval-ind* boundaries crossed between the first and second date. For example, the number of weeks between Sunday, January 4 and Sunday, January 11 is 1. The number of months between March 31 and April 1 would be 1 because the month boundary is crossed from March to April.

The TIMESTAMPDIFF function generates an error if the result is out of range for integer values. For seconds, the maximum number is equivalent to approximately 68 years. The TIMESTAMPDIFF function generates an error if a difference in weeks is requested and one of the two dates precedes January 7 of the year 0001.

Examples of TIMESTAMPDIFF

- This function returns the value `1` because a 1-second boundary is crossed even though the two timestamps differ by only one microsecond:

  ```
  TIMESTAMPDIFF (SQL_TSI_SECOND, TIMESTAMP '2006-09-12 11:59:58.999999',
  TIMESTAMP '2006-09-12 11:59:59.000000')
  ```

- This function returns the value `0` because no 1-second boundaries are crossed:

  ```
  TIMESTAMPDIFF (SQL_TSI_YEAR, TIMESTAMP '2006-12-31 23:59:59.00000,
  TIMESTAMP '2006-12-31 23:59:59.999999')
  ```

- This function returns the value `1` because a year boundary is crossed:

  ```
  TIMESTAMPDIFF (SQL_TSI_YEAR, TIMESTAMP '2006-12-31 23:59:59.999999',
  TIMESTAMP '2007-01-01 00:00:00.000000;)
  ```

- This function returns the value `1` because a WEEK boundary is crossed:

  ```
  TIMESTAMPDIFF (SQL_TSI_WEEK, DATE '2006-01-01', DATE '2006-01-09')
  ```

- This function returns the value of `-29`:

  ```
  TIMESTAMPDIFF (SQL_TSI_DAY, DATE '2004-03-01', DATE '2004-02-01')
  ```

TRANSLATE Function

The TRANSLATE function translates a character string from a source character set to a target character set. The TRANSLATE function changes both the character string data type and the character set encoding of the string.

```
TRANSLATE(character-value-expression USING translation-name)
```

character-value-expression

is a character string.

translation-name

is one of these translation names:

Translation Name	Source Character Set	Target Character Set	Comments
ISO88591TOUTF8	ISO88591	UTF8	Translates ISO8859-1 characters to UTF8 characters. No data loss is possible.
UTF8TOISO88591	UTF8	ISO88591	Translates UTF8 characters to ISO88591 characters. Trafodion SQL will display an error if it encounters a Unicode character that cannot be converted to the target character set.

translation-name identifies the translation, source and target character set. When you translate to the UTF8 character set, no data loss is possible. However, when Trafodion SQL translates a *character-value-expression* from UTF8, it may be that certain characters cannot be converted to the target character set. Trafodion SQL reports an error in this case.

Trafodion SQL returns a variable-length character string with character repertoire equal to the character repertoire of the target character set of the translation and the maximum length equal to the fixed length or maximum variable length of the source *character-value-expression*.

If you enter an illegal *translation-name*, Trafodion SQL returns an error.

If the character set for *character-value-expression* is different from the source character set as specified in the *translation-name*, Trafodion SQL returns an error.

TRIM Function

The TRIM function removes leading and trailing characters from a character string. Every character, including multibyte characters, is treated as one character.

```
TRIM ([[trim-type] [trim-char] FROM] trim-source)

trim-type is:
   LEADING | TRAILING | BOTH
```

trim-type

specifies whether characters are to be trimmed from the leading end (LEADING), trailing end (TRAILING), or both ends (BOTH) of *trim-source*. If you omit *trim-type*, the default is BOTH.

trim_char

is an SQL character value expression and specifies the character to be trimmed from *trim-source*. *trim_char* has a maximum length of 1. If you omit *trim_char*, SQL trims blanks (' ') from *trim-source*.

trim-source

is an SQL character value expression and specifies the string from which to trim characters. See "Character Value Expressions" (page 211).

Considerations for TRIM

Result of TRIM

The result is always of type VARCHAR, with maximum length equal to the fixed length or maximum variable length of *trim-source*. If the source character string is an upshifts CHAR or VARCHAR string, the result is an upshifts VARCHAR type.

Examples of TRIM

- Return 'Robert':

  ```
  TRIM ('    Robert    ')
  ```

- The EMPLOYEE table defines FIRST_NAME as CHAR(15) and LAST_NAME as CHAR(20). This expression uses the TRIM function to return the value 'Robert Smith' without extra blanks:

  ```
  TRIM (first_name) || ' ' || TRIM (last_name)
  ```

UCASE Function

- "Considerations for UCASE"
- "Examples of UCASE"

The UCASE function upshifts alphanumeric characters. For non-alphanumeric characters, UCASE returns the same character. UCASE can appear anywhere in a query where a value can be used, such as in a select list, an ON clause, a WHERE clause, a HAVING clause, a LIKE predicate, an expression, or as qualifying a new value in an UPDATE or INSERT statement. The result returned by the UCASE function is equal to the result returned by the "UPPER Function" (page 423) or "UPSHIFT Function" (page 424).

UCASE returns a string of fixed-length or variable-length character data, depending on the data type of the input string.

UCASE is a Trafodion SQL extension.

```
UCASE (character-expression)
```

character-expression

is an SQL character value expression that specifies a string of characters to upshift. See "Character Value Expressions" (page 211).

Considerations for UCASE

For a UTF8 character expression, the UCASE function upshifts all lowercase or title case characters to uppercase and returns a character string. If the argument is of type CHAR(n) or VARCHAR(n), the result is of type VARCHAR(min(3n, 2048)), where the maximum length of VARCHAR is the minimum of 3n or 2048, whichever is smaller.

A lowercase character is a character that has the "alphabetic" property in Unicode Standard 2 and whose Unicode name includes lower. An uppercase character is a character that has the "alphabetic" property and whose Unicode name includes upper. A title case character is a character that has the Unicode "alphabetic" property and whose Unicode name includes *title*.

Examples of UCASE

Suppose that your CUSTOMER table includes an entry for Hotel Oregon. Select the column CUSTNAME and return in uppercase and lowercase letters by using the UCASE and LCASE functions:

```
SELECT custname,UCASE(custname),LCASE(custname)
FROM sales.customer;

(EXPR)              (EXPR)               (EXPR)
------------------  -------------------  ------------------
...                 ...                  ...
Hotel Oregon        HOTEL OREGON         hotel oregon

--- 17 row(s) selected.
```

See "LCASE Function" (page 352).

For more examples of when to use the UCASE function, see "UPSHIFT Function" (page 424).

UPPER Function

The UPPER function upshifts alphanumeric characters. For non-alphanumeric characters, UCASE returns the same character. UPPER can appear anywhere in a query where a value can be used, such as in a select list, an ON clause, a WHERE clause, a HAVING clause, a LIKE predicate, an expression, or as qualifying a new value in an UPDATE or INSERT statement. The result returned by the UPPER function is equal to the result returned by the "UPSHIFT Function" (page 424) or "UCASE Function" (page 422).

UPPER returns a string of fixed-length or variable-length character data, depending on the data type of the input string.

```
UPPER (character-expression)
```

character-expression

is an SQL character value expression that specifies a string of characters to upshift. See "Character Value Expressions" (page 211).

Example of UPPER

Suppose that your CUSTOMER table includes an entry for Hotel Oregon. Select the column CUSTNAME and return in uppercase and lowercase letters by using the UPPER and LOWER functions:

```
SELECT custname,UPPER(custname),LOWER(custname)
FROM sales.customer;

(EXPR)               (EXPR)               (EXPR)
-----------------  -------------------  ------------------
...                ...                  ...
Hotel Oregon       HOTEL OREGON         hotel oregon

--- 17 row(s) selected.
```

See "LOWER Function" (page 357).

For examples of when to use the UPPER function, see "UPSHIFT Function" (page 424).

UPSHIFT Function

The UPSHIFT function upshifts alphanumeric characters. For non-alphanumeric characters, UCASE returns the same character. UPSHIFT can appear anywhere in a query where a value can be used, such as in a select list, an ON clause, a WHERE clause, a HAVING clause, a LIKE predicate, an expression, or as qualifying a new value in an UPDATE or INSERT statement. The result returned by the UPSHIFT function is equal to the result returned by the "UPPER Function" (page 423) or "UCASE Function" (page 422).

UPSHIFT returns a string of fixed-length or variable-length character data, depending on the data type of the input string.

UPSHIFT is a Trafodion SQL extension.

```
UPSHIFT (character-expression)
```

character-expression

is an SQL character value expression that specifies a string of characters to upshift. See "Character Value Expressions" (page 211).

Examples of UPSHIFT

- Suppose that your CUSTOMER table includes an entry for Hotel Oregon. Select the column CUSTNAME and return a result in uppercase and lowercase letters by using the UPSHIFT, UPPER, and LOWER functions:

```
SELECT UPSHIFT(custname),UPPER(custname),UCASE(custname)
FROM sales.customer;

(EXPR)              (EXPR)               (EXPR)
------------------  -------------------  ------------------
...                 ...                  ...
HOTEL OREGON        HOTEL OREGON         HOTEL OREGON

--- 17 row(s) selected.
```

- Perform a case-insensitive search for the DataSpeed customer:

```
SELECT *
FROM sales.customer
WHERE UPSHIFT (custname) = 'DATASPEED';

CUSTNUM  CUSTNAME    STREET                CITY       ...
-------  ----------  --------------------  ---------  ...
   1234  DataSpeed   300 SAN GABRIEL WAY   NEW YORK   ...

--- 1 row(s) selected.
```

  In the table, the name can be in lowercase, uppercase, or mixed case letters.

- Suppose that your database includes two department tables: DEPT1 and DEPT2. Return all rows from the two tables in which the department names have the same value regardless of case:

```
SELECT * FROM persnl.dept1 D1, persnl.dept2 D2
WHERE UPSHIFT(D1.deptname) = UPSHIFT(D2.deptname);
```

USER Function

The USER function returns either the database username associated with the specified user ID number or the database username of the current user who invoked the function. The current user is the authenticated user who started the session. That database username is used for authorization of SQL statements in the current session.

```
USER [(user-id)]
```

user-id

is the 32-bit number associated with a database username.

The USER function is similar to the "AUTHNAME Function" (page 292) and the "CURRENT_USER Function" (page 319).

Considerations for USER

- This function can be specified only in the top level of a SELECT statement.
- The value returned is string data type VARCHAR(128) and is in ISO8859-1 encoding.

Examples of USER

- This example shows the database username of the current user who is logged in to the session:

```
>>SELECT USER FROM (values(1)) x(a);
(EXPR)
------------------------
TSHAW
--- 1 row(s) selected.
```

- This example shows the database username associated with the user ID number, 33333:

```
>>SELECT USER (33333) FROM (values(1)) x(a);
(EXPR)
------------------------
DB__ROOT
--- 1 row(s) selected.
```

VARIANCE Function

- "Considerations for VARIANCE"
- "Examples of VARIANCE"

VARIANCE is an aggregate function that returns the statistical variance of a set of numbers.

VARIANCE is a Trafodion SQL extension.

```
VARIANCE ([ALL | DISTINCT] expression [,weight])
```

ALL | DISTINCT

specifies whether duplicate values are included in the computation of the VARIANCE of the *expression*. The default option is ALL, which causes duplicate values to be included. If you specify DISTINCT, duplicate values are eliminated before the VARIANCE function is applied. If DISTINCT is specified, you cannot specify *weight*.

expression

specifies a numeric value expression that determines the values for which to compute the variance. *expression* cannot contain an aggregate function or a subquery. The DISTINCT clause specifies that the VARIANCE function operates on distinct values from the one-column table derived from the evaluation of *expression*.

weight

specifies a numeric value expression that determines the weights of the values for which to compute the variance. *weight* cannot contain an aggregate function or a subquery. *weight* is defined on the same table as *expression*. The one-column table derived from the evaluation of *expression* and the one-column table derived from the evaluation of *weight* must have the same cardinality.

Considerations for VARIANCE

Definition of VARIANCE

Suppose that *vi* are the values in the one-column table derived from the evaluation of *expression*. *N* is the cardinality of this one-column table that is the result of applying the *expression* to each row of the source table and eliminating rows that are null.

If *weight* is specified, *wi* are the values derived from the evaluation of *weight*. *N* is the cardinality of the two-column table that is the result of applying the *expression* and *weight* to each row of the source table and eliminating rows that have nulls in either column.

Definition When Weight Is Not Specified

If *weight* is not specified, the statistical variance of the values in the one-column result table is defined as:

where *vi* is the i-th value of *expression*, *v* is the average value expressed in the common data type, and N is the cardinality of the result table.

Because the definition of variance has *N-1* in the denominator of the expression (when weight is not specified), Trafodion SQL returns a default value of zero (and no error) if the number of rows in the table, or a group of the table, is equal to 1.

Definition When Weight Is Specified

If *weight* is specified, the statistical variance of the values in the two-column result table is defined as:

where vi is the i-th value of *expression*, *wi* is the i-th value of *weight*, *vw* is the weighted average value expressed in the common data type, and N is the cardinality of the result table.

Weighted Average

The weighted average *vw* of *vi* and *wi* is defined as:

where vi is the i-th value of *expression*, *wi* is the i-th value of *weight*, and N is the cardinality of the result table.

Data Type of the Result

The data type of the result is always DOUBLE PRECISION.

Operands of the Expression

The expression includes columns from the rows of the SELECT result table—but cannot include an aggregate function. These expressions are valid:

```
VARIANCE (SALARY)
VARIANCE (SALARY * 1.1)
VARIANCE (PARTCOST * QTY_ORDERED)
```

Nulls

VARIANCE is evaluated after eliminating all nulls from the set. If the result table is empty, VARIANCE returns NULL.

FLOAT(54) and DOUBLE PRECISION Data

Avoid using large FLOAT(54) or DOUBLE PRECISION values as arguments to VARIANCE. If SUM(x * x) exceeds the value of 1.15792089237316192e77 during the computation of VARIANCE(x), then a numeric overflow occurs.

Examples of VARIANCE

- Compute the variance of the salary of the current employees:

```
SELECT VARIANCE(salary) AS Variance_Salary
FROM persnl.employee;

VARIANCE_SALARY
-------------------------
 1.27573263588496116E+009

--- 1 row(s) selected.
```

- Compute the variance of the cost of parts in the current inventory:

```
SELECT VARIANCE (price * qty_available)
FROM sales.parts;

(EXPR)
-------------------------
 5.09652410092950336E+013

--- 1 row(s) selected.
```

WEEK Function

The WEEK function converts a DATE or TIMESTAMP expression into an INTEGER value in the range 1 through 54 that represents the corresponding week of the year. If the year begins on a Sunday, the value `1` will be returned for any datetime that occurs in the first 7 days of the year. Otherwise, the value `1` will be returned for any datetime that occurs in the partial week before the start of the first Sunday of the year. The value `53` is returned for datetimes that occur in the last full or partial week of the year except for leap years that start on Saturday where December 31 is in the 54th full or partial week.

WEEK is a Trafodion SQL extension.

```
WEEK (datetime-expression)
```

datetime-expression

is an expression that evaluates to a datetime value of type DATE or TIMESTAMP. See "Datetime Value Expressions" (page 212).

Example of WEEK

Return an integer that represents the week of the year from the START_DATE column in the PROJECT table:

```
SELECT start_date, ship_timestamp, WEEK(start_date)
FROM persnl.project
WHERE projcode = 1000;

Start/Date  Time/Shipped                 (EXPR)
----------  --------------------------  --------------
2008-04-10  2008-04-21 08:15:00.000000              15
```

YEAR Function

The YEAR function converts a DATE or TIMESTAMP expression into an INTEGER value that represents the year.

YEAR is a Trafodion SQL extension.

```
YEAR (datetime-expression)
```

datetime-expression

is an expression that evaluates to a datetime value of type DATE or TIMESTAMP. See "Datetime Value Expressions" (page 212).

Example of YEAR

Return an integer that represents the year from the START_DATE column in the PROJECT table:

```
SELECT start_date, ship_timestamp, YEAR(start_date)
FROM persnl.project
WHERE projcode = 1000;

Start/Date  Time/Shipped                 (EXPR)
----------  --------------------------  ------
2008-04-10  2008-04-21 08:15:00.000000    2008
```

ZEROIFNULL Function

The ZEROIFNULL function returns a value of zero if the expression if NULL. Otherwise, it returns the value of the expression.

```
ZEROIFNULL (expression)
```

expression

specifies a value expression. It must be a numeric data type.

Example of ZEROIFNULL

`ZEROIFNULL` returns the value of the column named `salary` whenever the column value is not NULL and it returns 0 whenever the column value is NULL.

```
ZEROIFNULL (salary)
```

7 OLAP Functions

This section describes the syntax and semantics of the On Line Analytical Process (OLAP) window functions. The OLAP window functions are ANSI compliant.

Considerations for Window Functions

These considerations apply to all window functions.

inline-window-specification

The window defined by the *inline-window-specification* consists of the rows specified by the *window-frame-clause*, bounded by the current partition. If no PARTITION BY clause is specified, the partition is defined to be all the rows of the intermediate result. If a PARTITION BY clause is specified, the partition is the set of rows which have the same values for the expressions specified in the PARTITION clause.

window-frame-clause

DISTINCT is not supported for window functions.

Use of a FOLLOWING term is not supported. Using a FOLLOWING term results in an error.

If no *window-frame-clause* is specified, "ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING" is assumed. This clause is not supported because it involves a FOLLOWING term and will result in an error.

"ROWS CURRENT ROW" is equivalent to "ROWS BETWEEN CURRENT ROW AND CURRENT ROW".

"ROWS *preceding-row*" is equivalent to "ROWS BETWEEN *preceding-row* AND CURRENT ROW".

Nulls

All nulls are eliminated before the function is applied to the set of values. If the window contains all NULL values, the result of the window function is NULL.

If the specified window for a particular row consists of rows that are all before the first row of the partition (no rows in the window), the result of the window function is NULL.

ORDER BY Clause Supports Expressions For OLAP Functions

The ORDER BY clause of the OLAP functions now supports expressions. However, use of multiple OLAP functions with different expressions in the same query is not supported. The following examples show how expressions may be used in the ORDER BY clause.

```
SELECT -1 * annualsalary neg_total,
RANK() OVER (ORDER BY -1 * annualsalary) olap_rank
FROM employee;
```

Using an aggregate in the ORDER BY clause:

```
SELECT num,
RANK() OVER (ORDER BY SUM(annualsalary)) olap_rank
FROM employee
GROUP BY num;
```

Using multiple functions with the same expression in the ORDER BY clause:

```
SELECT num, workgroupnum,
  RANK() OVER (ORDER BY SUM (annualsalary)*num) olap_rank,
  DENSE_RANK() OVER (ORDER BY SUM (annualsalary)*num) olap_drank
  ROW_NUMBER() OVER (ORDER BY SUM (annualsalary)*num) olap_mum
FROM employee
GROUP BY num, workgroupnum, annualsalary;
```

Using more functions with the same expression in the ORDER BY clause:

```
SELECT num, workgroupnum, annualsalary,
  SUM(AnnualSalary) OVER (ORDER BY SUM(annualsalary)*num ROWS
UNBOUNDED PRECEDING),
  AVG(AnnualSalary) OVER (ORDER BY SUM(annualsalary)*num ROWS UNBOUNDED
PRECEDING),
  MIN(AnnualSalary) OVER (ORDER BY SUM(annualsalary)*num ROWS UNBOUNDED
PRECEDING),
  MAX(AnnualSalary) OVER (ORDER BY SUM(annualsalary)*num ROWS UNBOUNDED
PRECEDING),
  VARIANCE(AnnualSalary) OVER (ORDER BY SUM(annualsalary)*num ROWS
UNBOUNDED PRECEDING),
  STDDEV(AnnualSalary) OVER (ORDER BY SUM(annualsalary)*num ROWS
UNBOUNDED PRECEDING),
  COUNT(AnnualSalary) OVER (ORDER BY SUM(annualsalary)*num ROWS
UNBOUNDED PRECEDING),
FROM employee
GROUP BY num, workgroupnum, annualsalary;
```

Limitations for Window Functions

These limitations apply to all window functions.

- The ANSI *window-clause* is not supported by Trafodion. Only the *inline-window-specification* is supported. An attempt to use an ANSI *window-clause* will result in a syntax error.
- The *window-frame-clause* cannot contain a FOLLOWING term, either explicitly or implicitly. Because the default window frame clause contains an implicit FOLLOWING ("ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING"), the default is not supported. So, practically, the *window-frame-clause* is not optional. An attempt to use a FOLLOWING term, either explicitly or implicitly will result in the "4343" error message.
- The window frame units can only be ROWS. RANGE is not supported by Trafodion. An attempt to use RANGE will result in a syntax error.
- The ANSI *window-frame-exclusion-specification* is not supported by Trafodion. An attempt to use a *window-frame-exclusion-specification* will result in a syntax error.
- Multiple *inline-window-specifications* in a single SELECT clause are not supported. For each window function within a SELECT clause, the ORDER BY clause and PARTITION BY specifications must be identical. The window frame can vary within a SELECT clause. An attempt to use multiple *inline-window-specifications* in a single SELECT clause will result in the "4340" error message.
- The ANSI *null-ordering-specification* within the ORDER BY clause is not supported by Trafodion. Null values will always be sorted as if they are greater than all non-null values. This is slightly different than a null ordering of NULLS LAST. An attempt to use a *null-ordering-specification* will result in a syntax error.
- The ANSI *filter-clause* is not supported for window functions by Trafodion. The *filter-clause* applies to all aggregate functions (grouped and windowed) and that the *filter-clause* is not currently supported for grouped aggregate functions. An attempt to use a *filter-clause* will result in a syntax error.
- The DISTINCT value for the *set-qualifier-clause* within a window function is not supported. Only the ALL value is supported for the *set-qualifier-clause* within a window function. An attempt to use DISTINCT in a window function will result in the "4341" error message.

AVG Window Function

AVG is a window function that returns the average of nonnull values of the given expression for the current window specified by the *inline-window specification*.

```
AVG ([ALL] expression) OVER (inline-window-specification)

inline-window-specification is:
    [PARTITION BY expression [, expression]...]
    [ORDER BY expression [ASC[ENDING] | DESC[ENDING]]
              [,expression [ASC[ENDING] | DESC[ENDING]]]...]
    [ window-frame-clause ]

window-frame-clause is:
     ROWS CURRENT ROW
   | ROWS preceding-row
   | ROWS BETWEEN preceding-row AND preceding-row
   | ROWS BETWEEN preceding-row AND CURRENT ROW
   | ROWS BETWEEN preceding-row AND following-row
   | ROWS BETWEEN CURRENT ROW AND CURRENT ROW
   | ROWS BETWEEN CURRENT ROW AND following-row
   | ROWS BETWEEN following-row AND following-row

preceding-row is:
     UNBOUNDED PRECEDING
   | unsigned-integer PRECEDING

following-row is:
     UNBOUNDED FOLLOWING
   | unsigned-integer FOLLOWING
```

ALL

specifies whether duplicate values are included in the computation of the AVG of the *expression*. The default option is ALL, which causes duplicate values to be included.

expression

specifies a numeric or interval value *expression* that determines the values to average.

See "Numeric Value Expressions" (page 218) and "Interval Value Expressions" (page 215).

inline-window-specification

specifies the window over which the AVG is computed. The *inline-window-specification* can contain an optional PARTITION BY clause, an optional ORDER BY clause and an optional window frame clause. The PARTITION BY clause specifies how the intermediate result is partitioned and the ORDER BY clause specifies how the rows are ordered within each partition.

window-frame-clause

specifies the window within the partition over which the AVG is computed.

Examples of AVG Window Function

- Return the running average value of the SALARY column:

```
SELECT empnum, AVG(salary)
                OVER (ORDER BY empnum ROWS UNBOUNDED PRECEDING)
FROM persnl.employee;
```

- Return the running average value of the SALARY column within each department:

```
SELECT deptnum, empnum, AVG(salary)
   OVER (PARTITION BY deptnum ORDER BY empnum ROWS UNBOUNDED PRECEDING)
FROM persnl.employee;
```

- Return the moving average of salary within each department over a window of the last 4 rows:

```
SELECT deptnum, empnum, AVG(SALARY)
   OVER (PARTITION BY deptnum ORDER BY empnum ROWS 3 PRECEDING)
FROM persnl.employee;
```

COUNT Window Function

COUNT is a window function that returns the count of the non null values of the given expression for the current window specified by the inline-window-specification.

```
COUNT  {(*) | ([ALL] expression) } OVER
inline-window-specification

inline-window-specification is:
    [PARTITION BY expression [, expression]...]
    [ORDER BY expression [ASC[ENDING] | DESC[ENDING]]
              [,expression [ASC[ENDING] | DESC[ENDING]]]...]
    [ window-frame-clause ]

window-frame-clause is:
     ROW CURRENT ROW
   | ROW preceding-row
   | ROW BETWEEN preceding-row AND preceding-row
   | ROW BETWEEN preceding-row AND CURRENT ROW
   | ROW BETWEEN preceding-row AND following-row
   | ROW BETWEEN CURRENT ROW AND CURRENT ROW
   | ROW BETWEEN CURRENT ROW AND following-row
   | ROW BETWEEN following-row AND following-row

preceding-row is:
     UNBOUNDED PRECEDING
   | unsigned-integer PRECEDING

following-row is:
     UNBOUNDED FOLLOWING
   | unsigned-integer FOLLOWING
```

ALL

specifies whether duplicate values are included in the computation of the COUNT of the *expression*. The default option is ALL, which causes duplicate values to be included.

expression

specifies a value *expression* that is to be counted. See "Expressions" (page 211).

inline-window-specification

specifies the window over which the COUNT is computed. The *inline-window-specification* can contain an optional PARTITION BY clause, an optional ORDER BY clause and an optional window frame clause. The PARTITION BY clause specifies how the intermediate result is partitioned and the ORDER BY clause specifies how the rows are ordered within each partition.

window-frame-clause

specifies the window within the partition over which the COUNT is computed.

Examples of COUNT Window Function

- Return the running count of the SALARY column:

```
SELECT empnum, COUNT(salary)
      OVER (ORDER BY empnum ROWS UNBOUNDED PRECEDING)
FROM persnl.employee;
```

- Return the running count of the SALARY column within each department:

```
SELECT deptnum, empnum, COUNT(salary)
     OVER (PARTITION BY deptnum ORDER BY empnum ROWS UNBOUNDED PRECEDING)
FROM persnl.employee;
```

- Return the moving count of salary within each department over a window of the last 4 rows:

```
SELECT deptnum, empnum, COUNT(salary)
      OVER (PARTITION BY deptnum ORDER BY empnum ROWS 3 PRECEDING)
FROM persnl.employee;
```

- Return the running count of employees within each department:

```
SELECT deptnum, empnum, COUNT(*)
      OVER (PARTITION BY deptnum ORDER BY empnum ROWS UNBOUNDED PRECEDING)
FROM persnl.employee;
```

DENSE_RANK Window Funtion

DENSE_RANK is a window function that returns the ranking of each row of the current partition specified by the inline-window-specification. The ranking is relative to the ordering specified in the inline-window-specification. The return value of DENSE_RANK starts at 1 for the first row of the window. Values of the given expression that are equal have the same rank. The value of DENSE_RANK advances 1 when the value of the given expression changes.

```
DENSE_RANK() OVER (inline-window-specification)

inline-window-specification is:
    [PARTITION BY expression [, expression]...]
    [ORDER BY expression [ASC[ENDING] | DESC[ENDING]]
              [,expression [ASC[ENDING] | DESC[ENDING]]]...]
```

inline-window-specification

specifies the window over which the DENSE_RANK is computed. The *inline-window-specification* can contain an optional PARTITION BY clause and an optional ORDER BY clause. The PARTITION BY clause specifies how the intermediate result is partitioned and the ORDER BY clause specifies how the rows are ordered within each partition.

Examples of DENSE_RANK Window Function

- Return the dense rank for each employee based on employee number:

```
SELECT DENSE_RANK() OVER (ORDER BY empnum), *
FROM persnl.employee;
```

- Return the dense rank for each employee within each department based on salary:

```
SELECT DENSE_RANK() OVER (PARTITION BY deptnum ORDER BY salary), *
FROM persnl.employee;
```

MAX Window Function

MAX is a window function that returns the maximum value of all non null values of the given expression for the current window specified by the inline-window-specification.

```
MAX ([ALL] expression) OVER (inline-window-specification)

inline-window-specification is:
    [PARTITION BY expression [, expression]...]
    [ORDER BY expression [ASC[ENDING] | DESC[ENDING]]
              [,expression [ASC[ENDING] | DESC[ENDING]]]...]
    [ window-frame-clause ]

window-frame-clause is:
     ROWS CURRENT ROW
   | ROWS preceding-row
   | ROWS BETWEEN preceding-row AND preceding-row
   | ROWS BETWEEN preceding-row AND CURRENT ROW
   | ROWS BETWEEN preceding-row AND following-row
   | ROWS BETWEEN CURRENT ROW AND CURRENT ROW
   | ROWS BETWEEN CURRENT ROW AND following-row
```

```
   | ROWS BETWEEN following-row AND following-row

preceding-row is:
     UNBOUNDED PRECEDING
   | unsigned-integer PRECEDING

following-row is:
     UNBOUNDED FOLLOWING
   | unsigned-integer FOLLOWING
```

ALL
: specifies whether duplicate values are included in the computation of the MAX of the *expression*. The default option is ALL, which causes duplicate values to be included.

expression
: specifies an expression that determines the values over which the MAX is computed.

See "Expressions" (page 211).

inline-window-specification
: specifies the window over which the MAX is computed. The *inline-window-specification* can contain an optional PARTITION BY clause, an optional ORDER BY clause and an optional window frame clause. The PARTITION BY clause specifies how the intermediate result is partitioned and the ORDER BY clause specifies how the rows are ordered within each partition.

window-frame-clause
: specifies the window within the partition over which the MAX is computed.

Examples of MAX Window Function

- Return the running maximum of the SALARY column:

```
SELECT empnum, MAX(salary)
     OVER (ORDER BY empnum ROWS UNBOUNDED PRECEDING)
FROM persnl.employee;
```

- Return the running maximum of the SALARY column within each department:

```
SELECT deptnum, empnum, MAX(salary)
     OVER (PARTITION BY deptnum ORDER BY empnum ROWS UNBOUNDED PRECEDING)
FROM persnl.employee;
```

- Return the moving maximum of salary within each department over a window of the last 4 rows:

```
SELECT deptnum, empnum, MAX(salary)
     OVER (PARTITION BY deptnum ORDER BY empnum ROWS 3 PRECEDING)
FROM persnl.employee;
```

MIN Window Function

MIN is a window function that returns the minimum value of all non null values of the given expression for the current window specified by the inline-window-specification.

```
MIN ([ALL] expression) OVER (inline-window-specification)

inline-window-specification is:
    [PARTITION BY expression [, expression]...]
    [ORDER BY expression [ASC[ENDING] | DESC[ENDING]]
              [,expression [ASC[ENDING] | DESC[ENDING]]]...]
    [ window-frame-clause ]

window-frame-clause is:
     ROWS CURRENT ROW
   | ROWS preceding-row
   | ROWS BETWEEN preceding-row AND preceding-row
```

```
    | ROWS BETWEEN preceding-row AND CURRENT ROW
    | ROWS BETWEEN preceding-row AND following-row
    | ROWS BETWEEN CURRENT ROW AND CURRENT ROW
    | ROWS BETWEEN CURRENT ROW AND following-row
    | ROWS BETWEEN following-row AND following-row

preceding-row is:
     UNBOUNDED PRECEDING
    | unsigned-integer PRECEDING

following-row is:
     UNBOUNDED FOLLOWING
    | unsigned-integer FOLLOWING
```

ALL
: specifies whether duplicate values are included in the computation of the MIN of the *expression*. The default option is ALL, which causes duplicate values to be included.

expression
: specifies an expression that determines the values over which the MIN is computed

See "Expressions" (page 211).

inline-window-specification
: specifies the window over which the MIN is computed. The *inline-window-specification* can contain an optional PARTITION BY clause, an optional ORDER BY clause and an optional window frame clause. The PARTITION BY clause specifies how the intermediate result is partitioned and the ORDER BY clause specifies how the rows are ordered within each partition.

window-frame-clause
: specifies the window within the partition over which the MIN is computed.

Examples of MIN Window Function

- Return the running minimum of the SALARY column:

```
SELECT empnum, MIN(salary)
     OVER (ORDER BY empnum ROWS UNBOUNDED PRECEDING)
FROM persnl.employee;
```

- Return the running minimum of the SALARY column within each department:

```
SELECT deptnum, empnum, MIN(salary)
     OVER (PARTITION BY deptnum ORDER BY empnum ROWS UNBOUNDED PRECEDING)
FROM persnl.employee;
```

- Return the moving minimum of salary within each department over a window of the last 4 rows:

```
SELECT deptnum, empnum, MIN(salary)
     OVER (PARTITION BY deptnum ORDER BY empnum ROWS 3 PRECEDING)
FROM persnl.employee;
```

RANK Window Function

RANK is a window function that returns the ranking of each row of the current partition specified by the inline-window-specification. The ranking is relative to the ordering specified in the inline-window-specification. The return value of RANK starts at 1 for the first row of the window. Values that are equal have the same rank. The value of RANK advances to the relative position of the row in the window when the value changes.

```
RANK() OVER (inline-window-specification)

inline-window-specification is:
    [PARTITION BY expression [, expression]...]
```

```
    [ORDER BY expression [ASC[ENDING] | DESC[ENDING]]
             [,expression [ASC[ENDING] | DESC[ENDING]]]...]
```

inline-window-specification

specifies the window over which the RANK is computed. The *inline-window-specification* can contain an optional PARTITION BY clause and an optional ORDER BY clause. The PARTITION BY clause specifies how the intermediate result is partitioned and the ORDER BY clause specifies how the rows are ordered within each partition.

Examples of RANK Window Function

- Return the rank for each employee based on employee number:

```
SELECT RANK() OVER (ORDER BY empnum), *
FROM persnl.employee;
```

- Return the rank for each employee within each department based on salary:

```
SELECT RANK() OVER (PARTITION BY deptnum ORDER BY salary), *
FROM persnl.employee;
```

ROW_NUMBER Window Function

ROW_NUMBER is a window function that returns the row number of each row of the current window specified by the inline-window-specification.

```
ROW_NUMBER () OVER (inline-window-specification)

inline-window-specification is:
    [PARTITION BY expression [, expression]...]
    [ORDER BY expression [ASC[ENDING] | DESC[ENDING]]
             [,expression [ASC[ENDING] | DESC[ENDING]]]...]
```

inline-window-specification

specifies the window over which the ROW_NUMBER is computed. The *inline-window-specification* can contain an optional PARTITION BY clause and an optional ORDER BY clause. The PARTITION BY clause specifies how the intermediate result is partitioned and the ORDER BY clause specifies how the rows are ordered within each partition.

window-frame-clause

specifies the window within the partition over which the ROW_NUMBER is computed.

Examples of ROW_NUMBER Window Function

- Return the row number for each row of the employee table:

```
SELECT ROW_NUMBER () OVER(ORDER BY empnum), *
FROM persnl.employee;
```

- Return the row number for each row within each department:

```
SELECT ROW_NUMBER () OVER(PARTITION BY deptnum ORDER BY empnum), *
FROM persnl.employee;
```

STDDEV Window Function

STDDEV is a window function that returns the standard deviation of non null values of the given expression for the current window specified by the inline-window-specification.

```
STDDEV ([ALL] expression) OVER (inline-window-specification)

inline-window-specification is:
    [PARTITION BY expression [, expression]...]
    [ORDER BY expression [ASC[ENDING] | DESC[ENDING]]
             [,expression [ASC[ENDING] | DESC[ENDING]]]...]
    [ window-frame-clause ]
```

```
window-frame-clause is:
     ROWS CURRENT ROW
   | ROWS preceding-row
   | ROWS BETWEEN preceding-row AND preceding-row
   | ROWS BETWEEN preceding-row AND CURRENT ROW
   | ROWS BETWEEN preceding-row AND following-row
   | ROWS BETWEEN CURRENT ROW AND CURRENT ROW
   | ROWS BETWEEN CURRENT ROW AND following-row
   | ROWS BETWEEN following-row AND following-row

preceding-row is:
     UNBOUNDED PRECEDING
   | unsigned-integer PRECEDING

following-row is:
     UNBOUNDED FOLLOWING
   | unsigned-integer FOLLOWING
```

ALL

specifies whether duplicate values are included in the computation of the STDDEV of the *expression*. The default option is ALL, which causes duplicate values to be included.

expression

specifies a numeric or interval value *expression* that determines the values over which STDDEV is computed.

inline-window-specification

specifies the window over which the STDDEV is computed. The *inline-window-specification* can contain an optional PARTITION BY clause, an optional ORDER BY clause and an optional window frame clause. The PARTITION BY clause specifies how the intermediate result is partitioned and the ORDER BY clause specifies how the rows are ordered within each partition.

window-frame-clause

specifies the window within the partition over which the STDDEV is computed.

Examples of STDDEV

- Return the standard deviation of the salary for each row of the employee table:

```
SELECT STDDEV(salary) OVER(ORDER BY empnum ROWS UNBOUNDED PRECEDING),*
FROM persnl.employee;
```

- Return the standard deviation for each row within each department:

```
SELECT STDDEV() OVER(PARTITION BY deptnum ORDER BY empnum ROWS UNBOUNDED PRECEDING), *
FROM persnl.employee;
```

SUM Window Function

SUM is a window function that returns the sum of non null values of the given expression for the current window specified by the inline-window-specification.

```
SUM ([ALL] expression) OVER (inline-window-specification)

inline-window-specification is:
    [PARTITION BY expression [, expression]...]
    [ORDER BY expression [ASC[ENDING] | DESC[ENDING]]
              [,expression [ASC[ENDING] | DESC[ENDING]]]...]
    [ window-frame-clause ]

window-frame-clause is:
     ROWS CURRENT ROW
   | ROWS preceding-row
   | ROWS BETWEEN preceding-row AND preceding-row
```

```
    | ROWS BETWEEN preceding-row AND CURRENT ROW
    | ROWS BETWEEN preceding-row AND following-row
    | ROWS BETWEEN CURRENT ROW AND CURRENT ROW
    | ROWS BETWEEN CURRENT ROW AND following-row
    | ROWS BETWEEN following-row AND following-row

preceding-row is:
     UNBOUNDED PRECEDING
    | unsigned-integer PRECEDING

following-row is:
     UNBOUNDED FOLLOWING
    | unsigned-integer FOLLOWING
```

ALL
: specifies whether duplicate values are included in the computation of the SUM of the *expression*. The default option is ALL, which causes duplicate values to be included.

expression
: specifies a numeric or interval value expression that determines the values to sum.

  See "Expressions" (page 211).

inline-window-specification
: specifies the window over which the SUM is computed. The *inline-window-specification* can contain an optional PARTITION BY clause, an optional ORDER BY clause and an optional window frame clause. The PARTITION BY clause specifies how the intermediate result is partitioned and the ORDER BY clause specifies how the rows are ordered within each partition.

window-frame-clause
: specifies the window within the partition over which the SUM is computed.

Examples of SUM Window Function

- Return the running sum value of the SALARY column:

```
SELECT empnum, SUM (salary)
                  OVER (ORDER BY empnum ROWS UNBOUNDED PRECEDING)
FROM persnl.employee;
```

- Return the running sum of the SALARY column within each department:

```
SELECT deptnum, empnum, SUM (salary)
   OVER (PARTITION BY deptnum ORDER BY empnum ROWS UNBOUNDED PRECEDING)
FROM persnl.employee;
```

- Return the moving sum of the SALARY column within each department over a window of the last 4 rows:

```
SELECT deptnum, empnum, SUM (salary)
   OVER (PARTITION BY deptnum ORDER BY empnum ROWS 3 PRECEDING)
FROM persnl.employee;
```

VARIANCE Window Function

VARIANCE is a window function that returns the variance of non null values of the given expression for the current window specified by the inline-window-specification.

```
VARIANCE ([ALL] expression) OVER (inline-window-specification)

inline-window-specification is:
    [PARTITION BY expression [, expression]...]
    [ORDER BY expression [ASC[ENDING] | DESC[ENDING]]
              [,expression [ASC[ENDING] | DESC[ENDING]]]...]
    [ window-frame-clause ]
```

```
window-frame-clause is:
     ROWS CURRENT ROW
   | ROWS preceding-row
   | ROWS BETWEEN preceding-row AND preceding-row
   | ROWS BETWEEN preceding-row AND CURRENT ROW
   | ROWS BETWEEN preceding-row AND following-row
   | ROWS BETWEEN CURRENT ROW AND CURRENT ROW
   | ROWS BETWEEN CURRENT ROW AND following-row
   | ROWS BETWEEN following-row AND following-row

preceding-row is:
     UNBOUNDED PRECEDING
   | unsigned-integer PRECEDING

following-row is:
     UNBOUNDED FOLLOWING
   | unsigned-integer FOLLOWING
```

ALL

specifies whether duplicate values are included in the computation of the VARIANCE of the *expression*. The default option is ALL, which causes duplicate values to be included.

expression

specifies a numeric or interval value expression that determines the values over which the variance is computed.

See "Expressions" (page 211).

inline-window-specification

specifies the window over which the VARIANCE is computed. The *inline-window-specification* can contain an optional PARTITION BY clause, an optional ORDER BY clause and an optional window frame clause. The PARTITION BY clause specifies how the intermediate result is partitioned and the ORDER BY clause specifies how the rows are ordered within each partition.

window-frame-clause

specifies the window within the partition over which the VARIANCE is computed.

Examples of VARIANCE Window Function

- Return the variance of the SALARY column:

```
SELECT empnum, VARIANCE (salary)
               OVER (ORDER BY empnum ROWS UNBOUNDED PRECEDING)
FROM persnl.employee;
```

- Return the variance of the SALARY column within each department:

```
SELECT deptnum, empnum, VARIANCE (salary)
   OVER (PARTITION BY deptnum ORDER BY empnum ROWS UNBOUNDED PRECEDING)
FROM persnl.employee;
```

8 SQL Runtime Statistics

The Runtime Management System (RMS) shows the status of queries while they are running. RMS can service on-demand requests from the Trafodion Command Interface (TrafCI) to get statistics for a given query ID or for active queries in a given process. RMS also provides information about itself to determine the health of the RMS infrastructure.

RMS provides the summary statistics for each fragment instance and detailed statistics for each operator (TDB_ID) of a given active query. A query is considered active if either the compilation or execution is in progress. The variable_input column output is returned as a multiple value pair of the form `token=value`. For more information, see "Considerations For Obtaining Statistics For Each Fragment-Instance of an Active Query" (page 460).

RMS is enabled and available all the time.

This chapter includes:

- "Retrieving SQL Runtime Statistics" (page 443)
- "Displaying SQL Runtime Statistics" (page 447)
- "Gathering Statistics About RMS " (page 457)
- "Using the QUERYID_EXTRACT Function" (page 459)
- "Considerations For Obtaining Statistics For Each Fragment-Instance of an Active Query" (page 460)

PERTABLE and OPERATOR Statistics

The SQL database engine determines which type of statistics collection is appropriate for the query. The RMS infrastructure provides the runtime metrics about a query while a query is executing. You can identify queries that are using excessive resources, suspend a query to determine its impact on resources, and cancel a query, when necessary. PERTABLE statistics count rows and report rows estimated in the operators in the disk processes and time spent in the ESP processes. Although PERTABLE statistics can deduce when all the rows have been read from the disks, it is impossible to correctly assess the current state of the query.

Complex queries such as joins, sorts, and group result sets are often too large to fit into memory, so intermediate results must overflow to scratch files. These operators are called Big Memory Operators (BMOs). Because of the BMOs, RMS provides OPERATOR statistics, which provide a richer set of statistics so that the current state of a query can be determined at any time.

With OPERATOR statistics, all SQL operators are instrumented and the following statistics are collected:

- Node time spent in the operator
- Actual number of rows flowing to the parent operator
- Estimated number of rows flowing to the parent operator (estimated by the optimizer)
- Virtual memory used in the BMO
- Amount of data overflowed to scratch files and read back to the query

For more information, see "Displaying SQL Runtime Statistics" (page 447).

Adaptive Statistics Collection

The SQL database engine chooses the appropriate statistics collection type based on the type of query. By default, the SQL database engine statistics collection is OPERATOR statistics. You can view the statistics in different formats: PERTABLE, ACCUMULATED, PROGRESS, and DEFAULT. Statistics Collection is adaptive to ensure that sufficient statistics information is available without

causing any performance impact to the query's execution. For some queries, either no statistics or PERTABLE statistics are collected.

Query Type	Statistics Collection Type
OLT optimized queries	PERTABLE
Unique queries	PERTABLE
CQD	No statistics
SET commands	No statistics
EXPLAIN	No statistics
GET STATISTICS	No statistics
All other queries	DEFAULT

Retrieving SQL Runtime Statistics

Using the GET STATISTICS Command

The GET STATISTICS command shows statistical information for:

- A single query ID (QID)
- Active queries for a process ID (PID)
- RMS itself

A query is considered active if either compilation or execution is in progress. In the case of a SELECT statement, a query is in execution until the statement or result set is closed. Logically, a query is considered to be active when the compile end time is -1 and the compile start time is not -1, or when the execute end time is -1 and the execute start time is not -1.

Syntax of GET STATISTICS

```
GET STATISTICS FOR QID {query-id|CURRENT}[stats-view-type]}
                  |PID {process-name|[nodeid,pid]}[ACTIVE n][stats-view-type]
                  |RMS node-num|ALL[RESET]

stats-view-type is:

ACCUMULATED|PERTABLE|PROGRESS|DEFAULT
```

QID

Required keyword if requesting statistics for a specific query.

query-id

is the query ID. You must put the *query-id* in double quotes if the user name in the query ID contains lower case letters or if the user name contains a period.

NOTE: The *query-id* is a unique identifier for the SQL statement generated when the query is compiled (prepared). The *query-id* is visible for queries executed through certain TrafCI commands.

CURRENT

provides statistics for the most recently prepared or executed statement in the same session where you run the GET STATISTICS FOR QID CURRENT command. You must issue the GET STATISTICS FOR QID CURRENT command immediately after the PREPARE or EXECUTE statement.

PID

Required keyword if requesting statistics for an active query in a given process.

process-name

is the name of the process ID (PID) in the format: `$Znnn`. The process name can be for the master (MXOSRVR) or executor server process (ESP). If the process name corresponds to the ESP, the ACTIVE *n* query is just the *n*th query in that ESP and might not be the currently active query in the ESP.

ACTIVE *n*

describes which of the active queries for which RMS returns statistics. ACTIVE 1 is the default. ACTIVE 1 returns statistics for the first active query. ACTIVE 2 returns statistics for the second active query.

stats-view-type

sets the statistics view type to a different format. Statistics are collected at the operator level by default. For exceptions, see "Adaptive Statistics Collection" (page 442).

ACCUMULATED

causes the statistics to be displayed in an aggregated summary across all tables in the query.

PERTABLE

displays statistics for each table in the query. This is the default *stats-view-type* although statistics are collected at the operator level. If the collection occurs at a lower level due to Adaptive Statistics, the default is the lowered collection level. For more information, see "Adaptive Statistics Collection" (page 442)

PROGRESS

displays rows of information corresponding to each of the big memory operators (BMO) operators involved in the query, in addition to pertable *stats-view-type*. For more information about BMOs, see "PERTABLE and OPERATOR Statistics" (page 442).

DEFAULT

displays statistics in the same way as it is collected.

RMS

Required keyword if requesting statistics about RMS itself.

node-num

returns the statistics about the RMS infrastructure for a given node.

ALL

returns the statistics about the RMS infrastructure for every node in the cluster.

RESET

resets the cumulative RMS statistics counters.

Examples of GET STATISTICS

These examples show the runtime statistics that various GET STATISTICS commands return. For more information about the runtime statistics and RMS counters, see "Displaying SQL Runtime Statistics" (page 447).

- This GET STATISTICS command returns PERTABLE statistics for the most recently executed statement in the same session:

```
SQL>get statistics for qid current;
Qid                    MXID11008018370212168211672476672000000000030000_59_SQL_CUR_6
Compile Start Time     2011/03/30 07:29:15.332216
Compile End Time       2011/03/30 07:29:15.339467
Compile Elapsed Time               0:00:00.007251
Execute Start Time     2011/03/30 07:29:15.383077
Execute End Time       2011/03/30 07:29:15.470222
Execute Elapsed Time               0:00:00.087145
State                  CLOSE
Rows Affected          0
```

```
SQL Error Code           100
Stats Error Code         0
Query Type               SQL_SELECT_NON_UNIQUE
Estimated Accessed Rows  0
Estimated Used Rows      0
Parent Qid               NONE
Child Qid                NONE
Number of SQL Processes  1
Number of Cpus           1
Execution Priority       -1
Transaction Id           -1
Source String            SELECT
CUR_SERVICE,PLAN,TEXT,CUR_SCHEMA,RULE_NAME,APPL_NAME,SESSION_NAME,DSN_NAME,ROLE_NAME,DEFAULT_SCHEMA_ACCESS_ONLY
 FROM(VALUES(CAST('HP_DEFAULT_SERVICE' as VARCHAR(50)),CAST(0 AS INT),CAST(0 AS INT),CAST('NEO.USR' as
VARCHAR(260)),CAST('' as VARCHAR(
SQL Source Length        548
Rows Returned            1
First Row Returned Time  2011/03/30 07:29:15.469778
Last Error before AQR    0
Number of AQR retries    0
Delay before AQR         0
No. of times reclaimed   0
Stats Collection Type    OPERATOR_STATS
SQL Process Busy Time    0
UDR Process Busy Time    0
SQL Space Allocated      32                      KB
SQL Space Used           3                       KB
SQL Heap Allocated       7                       KB
SQL Heap Used            1                       KB
EID Space Allocated      0                       KB
EID Space Used           0                       KB
EID Heap Allocated       0                       KB
EID Heap Used            0                       KB
Processes Created        0
Process Create Time      0
Request Message Count    0
Request Message Bytes    0
Reply Message Count      0
Reply Message Bytes      0
Scr. Overflow Mode       DISK
Scr File Count           0
Scr. Buffer Blk Size     0
Scr. Buffer Blks Read    0
Scr. Buffer Blks Written 0
Scr. Read Count          0
Scr. Write Count         0

--- SQL operation complete.
```

- This GET STATISTICS command returns PERTABLE statistics for the specified query ID (note that this command should be issued in the same session):

```
SQL>get statistics for qid "MXID11008005179212168187528072672000000000030000_48_SQL_CUR_2";
Qid                      MXID11008005179212168187528072672000000000030000_48_SQL_CUR_2
Compile Start Time       2011/03/30 00:53:21.382211
Compile End Time         2011/03/30 00:53:22.980201
Compile Elapsed Time              0:00:01.597990
Execute Start Time       2011/03/30 00:53:23.079979
Execute End Time         -1
Execute Elapsed Time              7:16:13.494563
State                    OPEN
Rows Affected            -1
SQL Error Code           0
Stats Error Code         0
Query Type               SQL_SELECT_NON_UNIQUE
Estimated Accessed Rows  2,487,984
Estimated Used Rows      2,487,984
Parent Qid               NONE
Child Qid                NONE
Number of SQL Processes  129
Number of Cpus           9
Execution Priority       -1
Transaction Id           34359956800
Source String            select count(*) from
MANAGEABILITY.INSTANCE_REPOSITORY.EVENTS_TEXT K,
MANAGEABILITY.INSTANCE_REPOSITORY.EVENTS_TEXT J,
MANAGEABILITY.INSTANCE_REPOSITORY.EVENTS_TEXT H,
MANAGEABILITY.INSTANCE_REPOSITORY.EVENTS_TEXT G
SQL Source Length        220
Rows Returned            0
First Row Returned Time  -1
Last Error before AQR    0
Number of AQR retries    0
Delay before AQR         0
No. of times reclaimed   0
Stats Collection Type    OPERATOR_STATS
SQL Process Busy Time    830,910,830,000
UDR Process Busy Time    0
SQL Space Allocated      179,049                 KB
SQL Space Used           171,746                 KB
SQL Heap Allocated       1,140,503               KB
SQL Heap Used            1,138,033               KB
```

```
EID Space Allocated      46,080                    KB
EID Space Used           42,816                    KB
EID Heap Allocated       18,624                    KB
EID Heap Used            192                       KB
Processes Created        32
Process Create Time      799,702
Request Message Count    202,214
Request Message Bytes    27,091,104
Reply Message Count      197,563
Reply Message Bytes      1,008,451,688
Scr. Overflow Mode       DISK
Scr File Count           0
Scr. Buffer Blk Size     0
Scr. Buffer Blks Read    0
Scr. Buffer Blks Written 0
Scr. Read Count          0
Scr. Write Count         0

Table Name
   Records Accessed       Records Used         Disk      Message             Message    Lock    Lock
  Disk Process    Open              Open
   Estimated/Actual   Estimated/Actual         I/Os        Count               Bytes    Escl    wait
     Busy Time   Count               Time
MANAGEABILITY.INSTANCE_REPOSITORY.EVENTS_TEXT(H)
            621,996            621,996
            621,998            621,998            0          441         10,666,384       0       0
       303,955      32             15,967
MANAGEABILITY.INSTANCE_REPOSITORY.EVENTS_TEXT(J)
            621,996            621,996
            621,998            621,998            0          439         10,666,384       0       0
       289,949      32             19,680
MANAGEABILITY.INSTANCE_REPOSITORY.EVENTS_TEXT(K)
            621,996            621,996
            621,998            621,998            0          439         10,666,384       0       0
       301,956      32             14,419
MANAGEABILITY.INSTANCE_REPOSITORY.EVENTS_TEXT(G)
                  0            621,996
                  0                  0            0          192          4,548,048       0       0
             0      32             40,019

--- SQL operation complete.
```

- This GET STATISTICS command returns ACCUMULATED statistics for the most recently executed statement in the same session:

```
SQL>get statistics for qid current accumulated;

Qid                      MXID11008025173212168212775343040000000000340000_957_SQL_CUR_6
Compile Start Time       2011/03/30 08:05:07.646667
Compile End Time         2011/03/30 08:05:07.647622
Compile Elapsed Time               0:00:00.000955
Execute Start Time       2011/03/30 08:05:07.652710
Execute End Time         2011/03/30 08:05:07.740461
Execute Elapsed Time               0:00:00.087751
State                    CLOSE
Rows Affected            0
SQL Error Code           100
Stats Error Code         0
Query Type               SQL_SELECT_NON_UNIQUE
Estimated Accessed Rows  0
Estimated Used Rows      0
Parent Qid               NONE
Child Qid                NONE
Number of SQL Processes  0
Number of Cpus           0
Execution Priority       -1
Transaction Id           -1
Source String            SELECT
CUR_SERVICE,PLAN,TEXT,CUR_SCHEMA,RULE_NAME,APPL_NAME,SESSION_NAME,DSN_NAME,ROLE_NAME,DEFAULT_SCHEMA_ACCESS_ONLY
 FROM(VALUES(CAST('HP_DEFAULT_SERVICE' as VARCHAR(50)),CAST(0 AS INT),CAST(0 AS INT),CAST('NEO.SCH' as
VARCHAR(260)),CAST('' as VARCHAR(
SQL Source Length        548
Rows Returned            1
First Row Returned Time  2011/03/30 08:05:07.739827
Last Error before AQR    0
Number of AQR retries    0
Delay before AQR         0
No. of times reclaimed   0
Stats Collection Type    OPERATOR_STATS
Accessed Rows            0
Used Rows                0
Message Count            0
Message Bytes            0
Stats Bytes              0
Disk IOs                 0
Lock Waits               0
Lock Escalations         0
Disk Process Busy Time   0
SQL Process Busy Time    0
UDR Process Busy Time    0
SQL Space Allocated      32                        KB
SQL Space Used           3                         KB
```

```
SQL Heap Allocated       7                          KB
SQL Heap Used            1                          KB
EID Space Allocated      0                          KB
EID Space Used           0                          KB
EID Heap Allocated       0                          KB
EID Heap Used            0                          KB
Opens                    0
Open Time                0
Processes Created        0
Process Create Time      0
Request Message Count    0
Request Message Bytes    0
Reply Message Count      0
Reply Message Bytes      0
Scr. Overflow Mode       UNKNOWN
Scr. File Count          0
Scr. Buffer Blk Size     0
Scr. Buffer Blks Read    0
Scr. Buffer Blks Written 0
Scr. Read Count          0
Scr. Write Count         0

--- SQL operation complete.
```

- These GET STATISTICS commands return PERTABLE statistics for the first active query in the specified process ID:

```
SQL>get statistics for pid 0,27195;
SQL>get statistics for pid $Z000F3R;
```

Displaying SQL Runtime Statistics

By default, GET STATISTICS displays table-wise statistics (PERTABLE). If you want to view the statistics in a different format, use the appropriate view option of the GET STATISTICS command.

RMS provides abbreviated statistics information for prepared statements and full runtime statistics for executed statements.

Table 3 (page 447) shows the RMS counters that are returned by GET STATISTICS, tokens from the STATISTICS table-valued function that relate to the RMS counters, and descriptions of the counters and tokens.

Table 3 RMS Counter Information for SQL Runtime Statistics

Counter Name	Tokens in STATISTICS Table-Valued Function	Description
Qid	Qid	A unique ID generated for each query. Each time a SQL statement is prepared, a new query ID is generated.
Compile Start Time	CompStartTime	Time when the query compilation started or time when PREPARE for this query started.
Compile End Time	CompEndTime	Time when the query compilation ended or time when PREPARE for this query ended.
Compile Elapsed Time	CompElapsedTime	Amount of actual time to prepare the query.
Execute Start Time	ExeStartTime	Time when query execution started.
Execute End Time	ExeEndTime	Time when query execution ended. When a query is executing, Execute End Time is –1.
Execute Elapsed Time	ExeElapsedTime	Amount of actual time used by the SQL executor to execute the query.
State	State	Internally used.
Rows Affected	RowsAffected	Represents the number of rows affected by the INSERT, UPDATE, or DELETE (IUD) SQL statements. Value of –1 for SELECT statements or non-IUD SQL statements.
SQL Error Code	SQLErrorCode	Top-level error code returned by the query, indicating whether the query completed with warnings, errors, or successfully. A positive number indicates a warning. A negative number indicates an

Table 3 RMS Counter Information for SQL Runtime Statistics *(continued)*

Counter Name	Tokens in STATISTICS Table-Valued Function	Description
		error. The value returned may not be accurate up to the point GET STATISTICS was executed.
Stats Error Code	StatsErrorCode	Error code returned to the statistics collector while obtaining statistics from RMS. If an error code, counter values may be incorrect. Reissue the GET STATISTICS command.
Query Type	QueryType	Type of DML statement and enum value: • SQL_SELECT_UNIQUE=1 • SQL_SELECT_NON_UNIQUE=2 • SQL_INSERT_UNIQUE=3 • SQL_INSERT_NON_UNIQUE=4 • SQL_UPDATE_UNIQUE=5 • SQL_UPDATE_NON_UNIQUE=6 • SQL_DELETE_UNIQUE=7 • SQL_DELETE_NON_UNIQUE=8 • SQL_CONTROL=9 • SQL_SET_TRANSACTION=10 • SQL_SET_CATALOG=11 • SQL_SET_SCHEMA=12 • SQL_CALL_NO_RESULT_SETS=13 • SQL_CALL_WITH_RESULT_SETS=14 • SQL_SP_RESULT_SET=15 • SQL_INSERT_ROWSET_SIDETREE=16 • SQL_CAT_UTIL=17 • SQL_EXE_UTIL=18 • SQL_OTHER=1 • SQL_UNKNOWN=0
Estimated Accessed Rows	EstRowsAccessed	Compiler's estimated number of rows accessed by the executor in TSE.
Estimated Used Rows	EstRowsUsed	Compiler's estimated number of rows returned by the executor in TSE after applying the predicates.
Parent Qid	parentQid	A unique ID for the parent query. If there is no parent query ID associated with the query, RMS returns NONE. For more information, see "Using the Parent Query ID" (page 455).
Child Qid	childQid	A unique ID for the child query. If there is no child query, then there will be no child query ID and RMS returns NONE. For more information, see "Child Query ID" (page 456).
Number of SQL Processes	numSqlProcs	Represents the number of SQL processes (excluding TSE processes) involved in executing the query.
Number of CPUs	numCpus	Represents the number of nodes that SQL is processing the query.
Transaction ID	transId	Represents the transaction ID of the transaction involved in executing the query. When no transaction exists, the Transaction ID is -1.
Source String	sqlSrc	Contains the first 254 bytes of source string.
SQL Source Length	sqlSrcLen	The actual length of the SQL source string.

Table 3 RMS Counter Information for SQL Runtime Statistics *(continued)*

Counter Name	Tokens in STATISTICS Table-Valued Function	Description
Rows Returned	rowsReturned	Represents the number of rows returned from the root operator at the master executor process.
First Row Returned Time	firstRowReturnTime	Represents the actual time that the first row is returned by the master root operator.
Last Error Before AQR	LastErrorBeforeAQR	The error code that triggered Automatic Query Retry (AQR) for the most recent retry. If the value is not 0, this is the error code that triggered the most recent AQR.
Number of AQR retries	AQRNumRetries	The number of retries for the current query until now.
Delay before AQR	DelayBeforeAQR	Delay in seconds that SQL waited before initiating AQR.
No. of times reclaimed	reclaimSpaceCnt	When a process is under virtual memory pressure, the execution space occupied by the queries executed much earlier will be reclaimed to free up space for the upcoming queries. This counter represents how many times this particular query is reclaimed.
	statsRowType	statsRowType can be one of the following: • SQLSTATS_DESC_OPER_STATS=0 • SQLSTATS_DESC_ROOT_OPER_STATS=1 • SQLSTATS_DESC_PERTABLE_STATS=11 • SQLSTATS_DESC_UDR_STATS=13 • SQLSTATS_DESC_MASTER_STATS=15 • SQLSTATS_DESC_RMS_STATS=16 • SQLSTATS_DESC_BMO_STATS=17
Stats Collection Type	StatsType	Collection type, which is OPERATOR_STATS by default. StatsType can be one of the following: • SQLCLI_NO_STATS=0 • SQLCLI_ACCUMULATED_STATS=2 • SQLCLI_PERTABLE_STATS=3 • SQLCLI_OPERATOR_STATS=5
Accessed Rows (Rows Accessed)	AccessedRows	Actual number of rows accessed by the executor in TSE.
Used Rows (Rows Used)	UsedRows	Number of rows returned by TSE after applying the predicates. In a push down plan, TSE may not return all the used rows.
Message Count	NumMessages	Count of the number of messages sent to TSE.
Message Bytes	MessageBytes	Count of the message bytes exchanged with TSE.
Stats Bytes	StatsBytes	Number of bytes returned for statistics counters from TSE.
Disk IOs	DiskIOs	Number of physical disk reads for accessing the tables.
Lock Waits	LockWaits	Number of times this statement had to wait on a conflicting lock.
Lock Escalations	Escalations	Number of times row locks escalated to a file lock during the execution of this statement.
Disk Process Busy Time	ProcessBusyTime	An approximation of the total node time in microseconds spent by TSE for executing the query.
SQL Process Busy Time	CpuTime	An approximation of the total node time in microseconds spent in the master and ESPs involved in the query.
UDR Process Busy Time (same as UDR CPU Time)	udrCpuTime	An approximation of the total node time in microseconds spent in the UDR server process.

Table 3 RMS Counter Information for SQL Runtime Statistics *(continued)*

Counter Name	Tokens in STATISTICS Table-Valued Function	Description
UDR Server ID	UDRServerId	MXUDR process ID.
Recent Request Timestamp		Actual timestamp of the recent request sent to MXUDR.
Recent Reply Timestamp		Actual timestamp of the recent request received by MXUDR.
SQL Space Allocated[1]	SpaceTotal[1]	The amount of "space" type of memory in KB allocated in the master and ESPs involved in the query.
SQL Space Used[1]	SpaceUsed[1]	Amount of "space" type of memory in KB used in master and ESPs involved in the query.
SQL Heap Allocated[2]	HeapTotal[2]	Amount of "heap" type of memory in KB allocated in master and ESPs involved in the query.
SQL Heap Used[2]	HeapUsed[2]	Amount of "heap" type of memory in KB used in master and ESPs involved in the query.
EID Space Allocated[1]	Dp2SpaceTotal	Amount of "space" type of memory in KB allocated in the executor in TSEs involved in the query.
EID Space Used[1]	Dp2SpaceUsed	Amount of "space" type of memory in KB used in the executor in TSEs involved in the query.
EID Heap Allocated[2]	Dp2HeapTotal	Amount of "heap" memory in KB allocated in the executor in TSEs involved in the query.
EID Heap Used[2]	Dp2HeapUsed	Amount of "heap" memory in KB used in the executor in TSEs involved in the query.
Opens	Opens	Number of OPEN calls performed by the SQL executor on behalf of this statement.
Open Time	OpenTime	Time (in microseconds) this process spent doing OPENs on behalf of this statement.
Processes Created	Newprocess	The number of processes (ESPs and MXCMPs) created by the master executor for this statement.
Process Create Time	NewprocessTime	The elapsed time taken to create these processes.
Table Name	AnsiName	Name of a table in the query.
Request Message Count	reqMsgCnt	Number of messages initiated from the master to ESPs or from the ESP to ESPs.
Request Message Bytes	regMsgBytes	Number of message bytes that are sent from the master to ESPs or from the ESP to ESPs as part of the request messages.
Reply Message Count	replyMsgCnt	Number of reply messages from the ESPs for the message requests.
Reply Message Bytes	replyMsgBytes	Number of bytes sent as part of the reply messages.
Scr. Overflow Mode	scrOverFlowMode	Represents the scratch overflow mode. Modes are DISK_TYPE or SSD_TYPE.
Scr. File Count	scrFileCount	Number of scratch files created to execute the query. Default file size is 2 GB.
Scr. Buffer Blk Size	scrBufferBlockSize	Size of buffer block that is used to read from/write to the scratch file.
Scr. Buffer Blks Read	scrBufferRead	Number of scratch buffer blocks read from the scratch file.
Scr. Buffer Blks Written	scrBufferWritten	Number of scratch buffer blocks written to the scratch file. Exact size of scratch file can be obtained by multiplying Scr. Buffer Blk Size by this counter.

Table 3 RMS Counter Information for SQL Runtime Statistics *(continued)*

Counter Name	Tokens in STATISTICS Table-Valued Function	Description
Scr. Read Count	scrReadCount	Number of file-system calls involved in reading buffer blocks from scratch files. One call reads multiple buffer blocks at once.
Scr. Write Count	scrWriteCount	Number of file-system calls involved in writing buffer blocks to scratch files. One call writes multiple buffer blocks at once.
BMO Heap Used	bmoHeapUsed	Amount of "heap" type of memory in KB used in the BMO operator(s). The BMO operators are HASH_JOIN (and all varieties of HASH_JOIN), HASH_GROUPBY (and all varieties of HASH_GROUPBY), and SORT (and all varieties of SORT).
BMO Heap Total	bmoHeapTotal	Amount of "heap" type of memory in KB allocated in the BMO operator(s).
BMO Heap High Watermark	bmoHeapWM	Maximum amount of memory used in the BMO operator.
BMO Space Buffer Size	bmoSpaceBufferSize	Size in KB for space buffers allocated for the type of memory.
BMO Space Buffer Count	bmoSpaceBufferCount	Count of space buffers allocated for the type of memory.
Records Accessed (Estimated / Actual)		Actual number of rows accessed by the executor in TSE.
Records Used (Estimated / Actual)		Number of rows returned by TSE after applying the predicates. In a push-down plan, TSE may not return all the used rows.
ID		TDB ID of the operator at the time of execution of the query.
LCID		Left child operator ID.
RCID		Right child operator ID.
PaID		Parent operator ID (TDB-ID).
ExID		Explain plan operator ID.
Frag		Fragment ID to which this operator belongs.
Dispatches		Number of times the operator is scheduled in SQL executor.
Oper CPU Time	OperCpuTime	Approximation of the node time spent by the operator to execute the query.
Est. Records Used		Approximation of the number of tuples that would flow up to the parent operator.
Act. Records Used		Actual number of tuples that flowed up to the parent operator.
	ProcessId	Name of the process ID (PID) in the format: $Znnn. The process name can be for the master (MXOSRVR) or executor server process (ESP).

[1] Space is memory allocated from a pool owned by the executor. The executor operators requesting the memory are not expected to return the memory until the statement is deallocated.

[2] Heap memory is used for temporary allocations. Operators may return heap memory before the statement is deallocated. This allows the memory to be reused as needed.

Examples of Displaying SQL Runtime Statistics

Statistics of a Prepared Statement

This example shows the output of the currently prepared statement:

```
SQL>get statistics for qid current;

Qid                    MXID11000006497212158373059979520000000001930000_4200_Q1
```

```
Compile Start Time       2010/12/06 10:55:40.931000
Compile End Time         2010/12/06 10:55:42.131845
Compile Elapsed Time                 0:00:01.200845
Execute Start Time       -1
Execute End Time         -1
Execute Elapsed Time                 0:00:00.000000
State                    CLOSE
Rows Affected            -1
SQL Error Code           0
Stats Error Code         0
Query Type               SQL_SELECT_NON_UNIQUE
Estimated Accessed Rows  100,010
Estimated Used Rows      100,010
Parent Qid               NONE
Child Qid                NONE
Number of SQL Processes  0
Number of Cpus           0
Execution Priority       -1
Transaction Id           -1
Source String            select * from t100k where b in (select b from t10)
SQL Source Length        50
Rows Returned            0
First Row Returned Time  -1
Last Error before AQR    0
Number of AQR retries    0
Delay before AQR         0
No. of times reclaimed   0
Stats Collection Type    OPERATOR_STATS

--- SQL operation complete.
```

PERTABLE Statistics of an Executing Statement

This example shows the PERTABLE statistics of an executing statement:

```
SQL>get statistics for qid current;

Qid                      MXID11000006497212158373059979520000000001930000_4200_Q1
Compile Start Time       2010/12/06 10:55:40.931000
Compile End Time         2010/12/06 10:55:42.131845
Compile Elapsed Time                 0:00:01.200845
Execute Start Time       2010/12/06 10:56:16.254686
Execute End Time         2010/12/06 10:56:18.434873
Execute Elapsed Time                 0:00:02.180187
State                    CLOSE
Rows Affected            0
SQL Error Code           100
Stats Error Code         0
Query Type               SQL_SELECT_NON_UNIQUE
Estimated Accessed Rows  100,010
Estimated Used Rows      100,010
Parent Qid               NONE
Child Qid                NONE
Number of SQL Processes  7
Number of Cpus           1
Execution Priority       -1
Transaction Id           18121
Source String            select * from t100k where b in (select b from t10)
SQL Source Length        50
Rows Returned            100
First Row Returned Time  2010/12/06 10:56:18.150977
Last Error before AQR    0
Number of AQR retries    0
Delay before AQR         0
No. of times reclaimed   0
Stats Collection Type    OPERATOR_STATS
SQL Process Busy Time    600,000
UDR Process Busy Time    0
SQL Space Allocated      1,576                    KB
SQL Space Used           1,450                    KB
SQL Heap Allocated       199                      KB
SQL Heap Used            30                       KB
EID Space Allocated      704                      KB
EID Space Used           549                      KB
EID Heap Allocated       582                      KB
EID Heap Used            6                        KB
Processes Created        4
Process Create Time      750,762
Request Message Count    701
Request Message Bytes    135,088
Reply Message Count      667
Reply Message Bytes      3,427,664
Scr. Overflow Mode       DISK
```

```
Scr File Count           0
Scr. Buffer Blk Size     0
Scr. Buffer Blks Read    0
Scr. Buffer Blks Written 0

Table Name
   Records Accessed       Records Used   Disk   Message   Message    Lock    Lock  Disk Process    Open    Open

   Estimated/Actual   Estimated/Actual   I/Os   Count     Bytes      Escl    wait  Busy Time       Count   Time
NEO.SCTEST.T10
                10                 10
                10                 10   0      2         5,280      0       0     2,000           0       0
NEO.SCTEST.T100K
           100,000            100,000
           100,000            100,000   0     110    3,235,720      0       0   351,941           4
48,747

--- SQL operation complete.
```

ACCUMULATED Statistics of an Executing Statement

This example shows the ACCUMULATED statistics of an executing statement:

```
SQL>get statistics for qid current accumulated;

Qid                        MXID11000006497212158373059979520000000001930000_4200_Q1
Compile Start Time         2010/12/06 10:55:40.931000
Compile End Time           2010/12/06 10:55:42.131845
Compile Elapsed Time                0:00:01.200845
Execute Start Time         2010/12/06 10:56:16.254686
Execute End Time           2010/12/06 10:56:18.434873
Execute Elapsed Time                0:00:02.180187
State                      CLOSE
Rows Affected              0
SQL Error Code             100
Stats Error Code           0
Query Type                 SQL_SELECT_NON_UNIQUE
Estimated Accessed Rows    100,010
Estimated Used Rows        100,010
Parent Qid                 NONE
Child Qid                  NONE
Number of SQL Processes    7
Number of Cpus             1
Execution Priority         -1
Transaction Id             18121
Source String              select * from t100k where b in (select b from t10)
SQL Source Length          50
Rows Returned              100
First Row Returned Time    2010/12/06 10:56:18.150977
Last Error before AQR      0
Number of AQR retries      0
Delay before AQR           0
No. of times reclaimed     0
Stats Collection Type      OPERATOR_STATS
Accessed Rows              100,010
Used Rows                  100,010
Message Count              112
Message Bytes              3,241,000
Stats Bytes                2,904
Disk IOs                   0
Lock Waits                 0
Lock Escalations           0
Disk Process Busy Time     353,941
SQL Process Busy Time      600,000
UDR Process Busy Time      0
SQL Space Allocated        1,576                      KB
SQL Space Used             1,450                      KB
SQL Heap Allocated         199                        KB
SQL Heap Used              30                         KB
EID Space Allocated        704                        KB
EID Space Used             549                        KB
EID Heap Allocated         582                        KB
EID Heap Used              6                          KB
Opens                      4
Open Time                  48,747
Processes Created          4
```

```
Process Create Time      750,762
Request Message Count    701
Request Message Bytes    135,088
Reply Message Count      667
Reply Message Bytes      3,427,664
Scr. Overflow Mode       DISK
Scr. File Count          0
Scr. Buffer Blk Size     0
Scr. Buffer Blks Read    0
Scr. Buffer Blks Written 0

--- SQL operation complete.
```

PROGRESS Statistics of an Executing Statement

This example shows the PROGRESS statistics of an executing statement:

```
SQL>get statistics for qid current PROGRESS;
Qid                      MXID11000006497212158373059979520000000001930000_4200_Q1
Compile Start Time       2010/12/06 10:55:40.931000
Compile End Time         2010/12/06 10:55:42.131845
Compile Elapsed Time                0:00:01.200845
Execute Start Time       2010/12/06 10:56:16.254686
Execute End Time         2010/12/06 10:56:18.434873
Execute Elapsed Time                0:00:02.180187
State                    CLOSE
Rows Affected            0
SQL Error Code           100
Stats Error Code         0
Query Type               SQL_SELECT_NON_UNIQUE
Estimated Accessed Rows  100,010
Estimated Used Rows      100,010
Parent Qid               NONE
Child Qid                NONE
Number of SQL Processes  7
Number of Cpus           1
Execution Priority       -1
Transaction Id           18121
Source String            select * from t100k where b in (select b from t10)
SQL Source Length        50
Rows Returned            100
First Row Returned Time  2010/12/06 10:56:18.150977
Last Error before AQR    0
Number of AQR retries    0
Delay before AQR         0
No. of times reclaimed   0
Stats Collection Type    OPERATOR_STATS
SQL Process Busy Time    600,000
SQL Space Allocated      1,576                    KB
SQL Space Used           1,450                    KB
SQL Heap Allocated       199                      KB
SQL Heap Used            30                       KB
EID Space Allocated      704                      KB
EID Space Used           549                      KB
EID Heap Allocated       582                      KB
EID Heap Used            6                        KB
Processes Created        4
Process Create Time      750,762
Request Message Count    701
Request Message Bytes    135,088
Reply Message Count      667
Reply Message Bytes      3,427,664

Table Name
   Records Accessed       Records Used         Disk      Message            Message    Lock    Lock       Disk
 Process    Open               Open
   Estimated/Actual   Estimated/Actual         I/Os        Count              Bytes    Escl    wait
Busy Time   Count               Time
NEO.SCTEST.T10
                10                 10
                10                 10            0            2              5,280       0       0
   2,000       0                  0
NEO.SCTEST.T100K
           100,000            100,000
           100,000            100,000            0          110          3,235,720       0       0
 351,941       4             48,747

   Id TDB Name                            Mode Phase            Phase Start Time
  BMO Heap Used BMO Heap Total    BMO Heap WM  BMO Space BufSz BMO Space BufCnt
  File Count       Scratch Buffer Block Size/Read/Written              Cpu Time
   16 EX_HASHJ                            DISK
            0              0                0               56                0
          0        -1                 0                   0             60,000
```

DEFAULT Statistics of an Executing Statement

This example shows the DEFAULT statistics of an executing statement:

```
SQL>get statistics for qid current DEFAULT;
Qid                     MXID11000006497212158373059979520000000001930000_4200_Q1
Compile Start Time      2010/12/06 10:55:40.931000
Compile End Time        2010/12/06 10:55:42.131845
Compile Elapsed Time               0:00:01.200845
Execute Start Time      2010/12/06 10:56:16.254686
Execute End Time        2010/12/06 10:56:18.434873
Execute Elapsed Time               0:00:02.180187
State                   CLOSE
Rows Affected           0
SQL Error Code          100
Stats Error Code        0
Query Type              SQL_SELECT_NON_UNIQUE
Estimated Accessed Rows 100,010
Estimated Used Rows     100,010
Parent Qid              NONE
Child Qid               NONE
Number of SQL Processes 7
Number of Cpus          1
Execution Priority      -1
Transaction Id          18121
Source String           select * from t100k where b in (select b from t10)
SQL Source Length       50
Rows Returned           100
First Row Returned Time 2010/12/06 10:56:18.150977
Last Error before AQR   0
Number of AQR retries   0
Delay before AQR        0
No. of times reclaimed  0
Stats Collection Type   OPERATOR_STATS

Id   LCId RCId PaId ExId Frag TDB Name           Dispatches     Oper CPU Time  Est. Records Used  Act. Records
 Used    Details

21   20   .    .    10   0    EX_ROOT                   15                 0                  0
     100
20   19   .    21   9    0    EX_SPLIT_TOP              13                 0                100
     100
19   18   .    20   9    0    EX_SEND_TOP               20                 0                100
     100
18   17   .    19   9    2    EX_SEND_BOTTOM            72                 0                100
     100
17   16   .    18   9    2    EX_SPLIT_BOTTOM           88                 0                100
     100
16   15   7    17   8    2    EX_HASHJ               1,341            60,000                100
     100
15   14   .    16   7    2    EX_SPLIT_TOP           1,343            20,000            100,000
 100,000
14   13   .    15   7    2    EX_SEND_TOP            1,343           120,000            100,000
 100,000
13   12   .    14   7    5    EX_SEND_BOTTOM         1,534           200,000            100,000
 100,000
12   11   .    13   7    5    EX_SPLIT_BOTTOM          493            70,000            100,000
 100,000
11   10   .    12   6    5    EX_SPLIT_TOP             486            70,000            100,000
 100,000
10   9    .    11   5    5    EX_PARTN_ACCESS        1,634            60,000            100,000
 100,000
9    8    .    10   5    6    EX_EID_ROOT              12                 0            100,000
       0
8    .    .    9    4    6    EX_DP2_SUBS_OPER         160           170,000            100,000
 100,000
7    6    .    16   3    2    EX_SPLIT_TOP             16                 0                 10
      10
6    5    .    7    3    2    EX_SEND_TOP              17                 0                 10
      10
5    4    .    6    3    3    EX_SEND_BOTTOM           17                 0                 10
      10
4    3    .    5    3    3    EX_SPLIT_BOTTOM           9                 0                 10
      10
3    2    .    4    2    3    EX_PARTN_ACCESS           6                 0                 10
      10
2    1    .    3    2    4    EX_EID_ROOT               3                 0                 10
       0
1    .    .    2    1    4    EX_DP2_SUBS_OPER          3            10,000                 10
      10

--- SQL operation complete.
```

Using the Parent Query ID

When executed, some SQL statements execute additional SQL statements, resulting in a parent-child relationship. For example, when executed, the UPDATE STATISTICS, MAINTAIN, and CALL statements execute other SQL statements called child queries. The child queries might execute even more child queries, thus introducing a hierarchy of SQL statements with parent-child relationships. The parent query ID maps the child query to the immediate parent SQL statement, helping you to trace the child SQL statement back to the user-issued SQL statement.

The parent query ID is available as a counter, `Parent Qid`, in the runtime statistics output. See Table 1-1 (page 11). A query directly issued by a user will not have a parent query ID and the counter will indicate "None."

Child Query ID

In many cases, a child query will execute in the same node as its parent. In such cases, the GET STATISTICS report on the parent query ID will contain a query ID value for the child query which executed most recently. Conversely, if no child query exists, or the child query is executing in a different node, no child query ID will be reported.

The following examples shows GET STATISTICS output for both the parent and one child query which are executed when the user issues a CREATE TABLE AS command:

```
SQL> -- get statistics for the parent query

SQL>get statistics for qid
+>MXID01001091200212164828759544076000000000217DEFAULT_MXCI_USER00_34___SQLCI_DML_LAST__
+>;
Qid                     MXID11001091200212164828759544076000000000217DEFAULT_MXCI_USER00_34___SQLCI_DML_LAST__
Compile Start Time      2011/02/18 14:49:04.606513
Compile End Time        2011/02/18 14:49:04.631802
Compile Elapsed Time              0:00:00.025289
Execute Start Time      2011/02/18 14:49:04.632142
Execute End Time        -1
Execute Elapsed Time              0:03:29.473604
State                   CLOSE
Rows Affected           -1
SQL Error Code          0
Stats Error Code        0
Query Type              SQL_INSERT_NON_UNIQUE
Estimated Accessed Rows 0
Estimated Used Rows     0
Parent Qid              NONE
Child Qid               MXID11001091200212164828759544076000000000217DEFAULT_MXCI_USER00_37_86
Number of SQL Processes 1
Number of Cpus          1
Execution Priority      148
Transaction Id          -1
Source String           create table odetail  hash partition by (ordernum, partnum) as select * from
SALES.ODETAIL;
SQL Source Length       91
Rows Returned           0
First Row Returned Time -1
Last Error before AQR   0
Number of AQR retries   0
Delay before AQR        0
No. of times reclaimed  0
Stats Collection Type   OPERATOR_STATS

Id   LCId RCId PaId ExId Frag TDB Name           Dispatches     Oper CPU Time  Est. Records Used  Act. Records
 Used    Details

2    1    .    .    2    0    EX_ROOT                      0                 0                  0
       0
1    .    .    2    1    0    CREATE_TABLE_AS              0                 0                  0
       0

--- SQL operation complete.

SQL> -- get statistics for the child query

SQL>get statistics for qid
+>MXID11001091200212164828759544076000000000217DEFAULT_MXCI_USER00_37_86
+>;
Qid                     MXID01001091200212164828759544076000000000217DEFAULT_MXCI_USER00_37_86
Compile Start Time      2011/02/18 14:49:07.632898
Compile End Time        2011/02/18 14:49:07.987334
Compile Elapsed Time              0:00:00.354436
Execute Start Time      2011/02/18 14:49:07.987539
Execute End Time        -1
Execute Elapsed Time              0:02:33.173486
State                   OPEN
Rows Affected           -1
SQL Error Code          0
Stats Error Code        0
Query Type              SQL_INSERT_NON_UNIQUE
Estimated Accessed Rows 101
Estimated Used Rows     101
Parent Qid              MXID10100109120021216482875954407600000000217DEFAULT_MXCI_USER00_34___SQLCI_DML_LAST__
Child Qid               NONE
Number of SQL Processes 1
Number of Cpus          1
Execution Priority      148
Transaction Id          \ARC0101(2).9.9114503
Source String           insert using sideinserts into CAT.SCH.ODETAIL  select * from SALES.ODETAIL;
```

```
SQL Source Length        75
Rows Returned            0
First Row Returned Time  -1
Last Error before AQR    0
Number of AQR retries    0
Delay before AQR         0
No. of times reclaimed   0
Stats Collection Type    OPERATOR_STATS

Id   LCId RCId PaId ExId Frag TDB Name           Dispatches     Oper CPU Time  Est. Records Used  Act. Records
 Used    Details

4    3    .    9    3    0    EX_SPLIT_TOP                1            10,062                100
       0
3    2    .    4    2    0    EX_PARTN_ACCESS            66             9,649                100
       0

--- SQL operation complete.
```

Gathering Statistics About RMS

Use the GET STATISTICS FOR RMS command to get information about RMS itself. The GET STATISTICS FOR RMS statement can be used to retrieve information about one node or all nodes. An individual report is provided for each node.

Counter	Description
CPU	The node number of the Trafodion cluster.
RMS Version	Internal version of RMS
SSCP PID	SQL Statistics control process ID.
SSCP Creation Timestamp	Actual timestamp when SQL statistics control process was created.
SSMP PID	SQL statistics merge process ID.
SSMP Creation Timestamp	Timestamp when SQL statistics merge was created.
Source String Store Len	Storage length of source string.
Stats Heap Allocated	Amount of memory allocated by all the queries executing in the given node in the RMS shared segments at this instance of time.
Stats Heap Used	Amount of memory used by all the queries executing in the given node in the RMS shared segment at this instance of time.
Stats Heap High WM	High amount of memory used by all the queries executing in the given node in the RMS shared segment until now.
No. of Process Regd.	Number of processes registered in the shared segment.
No. of Query Fragments Regd.	Number of query fragments registered in the shared segment.
RMS Semaphore Owner	Process ID that locked the semaphore at this instance of time.
No. of SSCPs Opened	Number of Statistics Control Processes opened. Normally, this should be equal to the number of nodes in the Trafodion cluster.
No. of SSCPs Open Deleted	Number of Statistics Control Processes with broken communication. Usually, this should be 0.
Last GC Time	The recent timestamp at which the shared segment was garbage collected.
Queries GCed in Last Run	Number of queries that were garbage collected in the recent GC run.

Counter	Description
Total Queries GCed	Total number of queries that were garbage collected since the statistics reset timestamp.
SSMP Request Message Count	Count of the number of messages sent from the SSMP process since the statistics reset timestamp.
SSMP Request Message Bytes	Number of messages bytes that are sent as part of the request from the SSMP process since the statistics reset timestamp.
SSMP Reply Message Count	Count of the number of reply messages received by the SSMP process since the statistics reset timestamp.
SSMP Reply Message Bytes	Number of messages bytes that are sent as part of the reply messages received by the SSMP process since the statistics reset timestamp.
SSCP Request Message Count	Count of the number of messages sent from the SSCP process since the statistics reset timestamp.
SSCP Request Message Bytes	Number of messages bytes are sent as part of the request from the SSCP process since the statistics reset timestamp.
SSCP Reply Message Count	Count of the number of reply messages received by the SSCP process since the statistics reset timestamp.
SSCP Reply Message Bytes	Number of messages bytes that are sent as part of the reply messages received by the SSCP process since the statistics reset timestamp.
RMS Stats Reset Timestamp	Timestamp for resetting RMS statistics.

```
SQL>get statistics for rms all;
Node name
CPU                             0
RMS Version                     2511
SSCP PID                        19521
SSCP Priority                   0
SSCP Creation Timestamp         2010/12/05 02:32:33.642752
SSMP PID                        19527
SSMP Priority                   0
SSMP Creation Timestamp         2010/12/05 02:32:33.893440
Source String Store Len         254
Stats Heap Allocated            0
Stats Heap Used                 3,002,416
Stats Heap High WM              3,298,976
No.of Process Regd.             157
No.of Query Fragments Regd.     296
RMS Semaphore Owner             -1
No.of SSCPs Opened              1
No.of SSCPs Open Deleted        0
Last GC Time                    2010/12/06 10:53:46.777432
Queries GCed in Last Run        55
Total Queries GCed              167
SSMP Request Message Count      58,071
SSMP Request Message Bytes      14,161,144
SSMP Reply Message Count        33,466
SSMP Reply Message Bytes        15,400,424
SSCP Request Message Count      3,737
SSCP Request Message Bytes      837,744
SSCP Reply Message Count        3,736
SSCP Reply Message Bytes        5,015,176
RMS Stats Reset Timestamp       2010/12/05 14:32:33.891083
```

```
--- SQL operation complete.
```

Using the QUERYID_EXTRACT Function

Use the QUERYID_EXTRACT function within an SQL statement to extract components of a query ID for use in a SQL query. The query ID, or QID, is a unique, clusterwide identifier for a query and is generated for dynamic SQL statements whenever a SQL string is prepared.

Syntax of QUERYID_EXTRACT

The syntax of the QUERYID_EXTRACT function is:

```
QUERYID_EXTRACT ('query-id', 'attribute')
```

query-id
is the query ID in string format.

attribute
is the attribute to be extracted. The value of *attribute* can be one of these parts of the query ID:

Attribute Value	Description
SEGMENTNUM	Logical node ID in Trafodion cluster
CPUNUM or CPU	Logical node ID in Trafodion cluster
PIN	Linux process ID number
EXESTARTTIME	Executor start time
SESSIONNUM	Session number
USERNAME	User name
SESSIONNAME	Session name
SESSIONID	Session ID
QUERYNUM	Query number
STMTNAME	Statement ID or handle

NOTE: The SEGMENTNUM and CPUNUM attributes are the same.

The result data type of the QUERYID_EXTRACT function is a VARCHAR with a length sufficient to hold the result. All values are returned in string format. Here is the QUERYID_EXTRACT function in a SELECT statement:

```
select queryid_extract('query-id', 'attribute-value') from (values(1)) as t1;
```

Examples of QUERYID_EXTRACT

- This command returns the node number of the query ID:

```
SQL>select substr(queryid_extract('MXID11000022675212170554548762240000000000206U6553500_21_S1', 'CPU'),1,
 20) from (values(1)) as t1;


(EXPR)
--------------------------------------------------------------------------------------------------------------------------------

0
```

```
--- 1 row(s) selected.
```

- This command returns the PIN of the query ID:

```
SQL>select substr(queryid_extract('MXID11000022675212170554548762240000000000206U6553500_21_S1', 'PIN'),1,
 20)  from (values(1)) as t1;


(EXPR)
--------------------------------------------------------------------------------------------------------------------------------

22675

--- 1 row(s) selected.
```

Statistics for Each Fragment-Instance of an Active Query

You can retrieve statistics for a query while it executes by using the STATISTICS table-valued function. Depending on the syntax used, you can obtain statistics summarizing each parallel fragment-instance of the query, or for any operator in each fragment-instance.

Syntax of STATISTICS Table-Valued Function

```
table(statistics (NULL, 'qid-str'))

qid-str is:
QID=query-id[,{TDBID_DETAIL=tdb-id|DETAIL=1}]
```

query-id
is the system-generated query ID. For example:

```
QID=MXID11000022675212170554548762240000000000206U6553500_21_S1
```

tdb-id
is the TDB ID of a given operator. TDB values can be obtained from the report returned from the GET STATISTICS command.

Considerations For Obtaining Statistics For Each Fragment-Instance of an Active Query

If the DETAIL=1 or TDBID_DETAIL=*tdb_id* options are used when the query is not executing, the STATISTICS table-valued function will not return any results.

The STATISTICS table-valued function can be used with a SELECT statement to return several columns. Many different counters exist in the *variable_info* column. The counters in this column are formatted as token-value pairs and the counters reported will depend on which option is used: DETAIL=1 or TDBID_DETAIL=*tdb_id*. If the TDBID_DETAIL option is used, the counters reported will also depend on the type of operator specified by the *tdb_id*. The reported counters can also be determined by the statsRowType counter.

For the counter descriptions, see Table 3 (page 447). The tokens for these counters are listed in the column “Tokens in STATISTICS Table-Valued Function”.

This query lists process names of all ESPs of an executing query identified by the given QID:

```
SQL>select
+>substr(variable_info,
+>      position('ProcessId:' in variable_info), 20) as processes
+>from
+>table(statistics(NULL,
+>'QID=MXID11000032684212170811581160672000000000206U6553500_19_S1,DETAIL=1' ))
+>group by 1;

PROCESSES
--------------------

ProcessId: $Z0000GS
ProcessId: $Z0000GT
```

```
ProcessId: $Z0000GU
ProcessId: $Z0000GV
ProcessId: $Z0102IQ
ProcessId: $Z000RNU
ProcessId: $Z0102IR
ProcessId: $Z0102IS
ProcessId: $Z0102IT

--- 9 row(s) selected.
```

This query gives BMO heap used for the hash join identified as TDB #15 in an executing query identified by the given QID:

```
SQL>select cast (
+>       substr(variable_info,
+>              position('bmoHeapUsed:' in variable_info),
+>              position('bmoHeapUsed:' in variable_info) +
+>              13 + (position(' ' in
+>                  substr(variable_info,
+>                         13 + position('bmoHeapUsed:' in variable_info))) -
+>                position('bmoHeapUsed:' in variable_info)))
+>        as char(25))
+>              from table(statistics(NULL,
+>'QID=MXID11000021706212170733911504160000000000206U6553500_25_S1,TDBID_DETAIL=15'));

(EXPR)
-------------------------

bmoHeapUsed: 3147
bmoHeapUsed: 3147
bmoHeapUsed: 3147
bmoHeapUsed: 3147
bmoHeapUsed: 3147
bmoHeapUsed: 3147
bmoHeapUsed: 3147
bmoHeapUsed: 3147

--- 8 row(s) selected.
```

A Reserved Words

The words listed in this appendix are reserved for use by Trafodion SQL. To prevent syntax errors, avoid using these words as identifiers in Trafodion SQL. In Trafodion SQL, if an operating system name contains a reserved word, you must enclose the reserved word in double quotes (") to access that column or object.

NOTE: In Trafodion SQL, ABSOLUTE, DATA, EVERY, INITIALIZE, OPERATION, PATH, SPACE, STATE, STATEMENT, STATIC, and START are not reserved words.

Reserved Trafodion SQL Identifiers

Trafodion SQL treats these words as reserved when they are part of Trafodion SQL stored text. They cannot be used as identifiers unless you enclose them in double quotes.

Table 4 Reserved SQL Identifiers — A

ACTION	ADD	ADMIN	AFTER	AGGREGATE
ALIAS	ALL	ALLOCATE	ALTER	AND
ANY	ARE	ARRAY	AS	ASC
ASSERTION	ASYNC	AT	AUTHORIZATION	AVG

Table 5 Reserved SQL Identifiers — B

BEFORE	BEGIN	BETWEEN	BINARY	BIT
BIT_LENGTH	BLOB	BOOLEAN	BOTH	BREADTH
BY				

Table 6 Reserved SQL Identifiers — C

CALL	CASCADE	CASCADED	CASE	CAST
CATALOG	CHAR	CHARACTER	CHARACTER_LENGTH	CHAR_LENGTH
CHECK	CLASS	CLOB	CLOSE	COALESCE
COLLATE	COLLATION	COLUMN	COMMIT	COMPLETION
CONNECT	CONNECTION	CONSTRAINT	CONSTRAINTS	CONSTRUCTOR
CONTINUE	CONVERT	CORRESPONDING	COUNT	CREATE
CROSS	CUBE	CURRENT	CURRENT_DATE	CURRENT_PATH
CURRENT_ROLE	CURRENT_TIME	CURRENT_TIMESTAMP	CURRENT_USER	CURRNT_USR_INTN
CURSOR	CYCLE			

Table 7 Reserved SQL Identifiers — D

DATE	DATETIME	DAY	DEALLOCATE	DEC
DECIMAL	DECLARE	DEFAULT	DEFERRABLE	DEFERRED
DELETE	DEPTH	DEREF	DESC	DESCRIBE
DESCRIPTOR	DESTROY	DESTRUCTOR	DETERMINISTIC	DIAGNOSTICS
DICTIONARY	DISCONNECT	DISTINCT	DOMAIN	DOUBLE
DROP	DYNAMIC			

Table 8 Reserved SQL Identifiers — E

EACH	ELSE	ELSEIF	END	END-EXEC
EQUALS	ESCAPE	EXCEPT	EXCEPTION	EXEC
EXECUTE	EXISTS	EXTERNAL	EXTRACT	

Table 9 Reserved SQL Identifiers — F

FALSE	FETCH	FIRST	FLOAT	FOR
FOREIGN	FOUND	FRACTION	FREE	FROM
FULL	FUNCTION			

Table 10 Reserved SQL Identifiers — G

GENERAL	GET	GLOBAL	GO	GOTO
GRANT	GROUP	GROUPING		

Table 11 Reserved SQL Identifiers — H

HAVING	HOST	HOUR		

Table 12 Reserved SQL Identifiers — I

IDENTITY	IF	IGNORE	IMMEDIATE	IN
INDICATOR	INITIALLY	INNER	INOUT	INPUT
INSENSITIVE	INSERT	INT	INTEGER	INTERSECT
INTERVAL	INTO	IS	ISOLATION	ITERATE

Table 13 Reserved SQL Identifiers — J

JOIN				

Table 14 Reserved SQL Identifiers — K

KEY				

Table 15 Reserved SQL Identifiers — L

LANGUAGE	LARGE	LAST	LATERAL	LEADING
LEAVE	LEFT	LESS	LEVEL	LIKE
LIMIT	LOCAL	LOCALTIME	LOCALTIMESTAMP	LOCATOR
LOOP	LOWER			

Table 16 Reserved SQL Identifiers — M

MAINTAIN	MAP	MATCH	MATCHED	MAX
MERGE	MIN	MINUTE	MODIFIES	MODIFY
MODULE	MONTH			

Table 17 Reserved SQL Identifiers — N

NAMES	NATIONAL	NATURAL	NCHAR	NCLOB
NEW	NEXT	NO	NONE	NOT
NULL	NULLIF	NUMERIC		

Table 18 Reserved SQL Identifiers — O

OCTET_LENGTH	OF	OFF	OID	OLD
ON	ONLY	OPEN	OPERATORS	OPTION
OPTIONS	OR	ORDER	ORDINALITY	OTHERS
OUT	OUTER	OUTPUT	OVERLAPS	

Table 19 Reserved SQL Identifiers — P

PAD	PARAMETER	PARAMETERS	PARTIAL	PENDANT
POSITION	POSTFIX	PRECISION	PREFIX	PREORDER
PREPARE	PRESERVE	PRIMARY	PRIOR	PRIVATE
PRIVILEGES	PROCEDURE	PROTECTED	PROTOTYPE	PUBLIC

Table 20 Reserved SQL Identifiers — Q

QUALIFY				

Table 21 Reserved SQL Identifiers — R

READ	READS	REAL	RECURSIVE	REF
REFERENCES	REFERENCING	RELATIVE	REORG	REORGANIZE
REPLACE	RESIGNAL	RESTRICT	RESULT	RETURN
RETURNS	REVOKE	RIGHT	ROLLBACK	ROLLUP
ROUTINE	ROW	ROWS		

Table 22 Reserved SQL Identifiers — S

SAVEPOINT	SCHEMA	SCOPE	SCROLL	SEARCH
SECOND	SECTION	SELECT	SENSITIVE	SESSION
SESSION_USER	SESSN_USR_INTN	SET	SETS	SIGNAL
SIMILAR	SIZE	SMALLINT	SOME	SPECIFIC
SPECIFICTYPE	SQL	SQL_CHAR	SQL_DATE	SQL_DECIMAL
SQL_DOUBLE	SQL_FLOAT	SQL_INT	SQL_INTEGER	SQL_REAL
SQL_SMALLINT	SQL_TIME	SQL_TIMESTAMP	SQL_VARCHAR	SQLCODE
SQLERROR	SQLEXCEPTION	SQLSTATE	SQLWARNING	STRUCTURE
SUBSTRING	SUM	SYNONYM	SYSTEM_USER	

Table 23 Reserved SQL Identifiers — T

TABLE	TEMPORARY	TERMINATE	TEST	THAN
THEN	THERE	TIME	TIMESTAMP	TIMEZONE_HOUR

Table 23 Reserved SQL Identifiers — T *(continued)*

TIMEZONE_MINUTE	TO	TRAILING	TRANSACTION	TRANSLATE
TRANSLATION	TRANSPOSE	TREAT	TRIGGER	TRIM
TRUE				

Table 24 Reserved SQL Identifiers — U

UNDER	UNION	UNIQUE	UNKNOWN	UNNEST
UPDATE	UPPER	UPSHIFT	USAGE	USER
USING				

Table 25 Reserved SQL Identifiers — V

VALUE	VALUES	VARCHAR	VARIABLE	VARYING
VIEW	VIRTUAL	VISIBLE		

Table 26 Reserved SQL Identifiers — W

WAIT	WHEN	WHENEVER	WHERE	WHILE
WITH	WITHOUT	WORK	WRITE	

Table 27 Reserved SQL Identifiers — Y

YEAR				

Table 28 Reserved SQL Identifiers — Z

ZONE				

B Control Query Default (CQD) Attributes

This appendix describes CQDs that are used to override system-level default settings.

HBase Environment CQDs

This section describes the CQD, "HBASE_INTERFACE" (page 466), which defines the HBase interface.

HBASE_INTERFACE

Category	HBase
Description	Interface to use to access HBase.
Values	Specify one of these values: • JNI to use a JNI interface • JNI_TRX to use a transactional interface with HBase-trx via JNI The default value is JNI_TRX.

Hive Environment CQDs

This section describes the CQD, "HIVE_MAX_STRING_LENGTH" (page 466), which defines the maximum string length for the `string` data type in Hive.

HIVE_MAX_STRING_LENGTH

Category	Hive
Description	Maximum supported string length for the `string` data type in Hive. All `string` columns in Hive tables get converted to VARCHAR(*n* BYTES) CHARACTER SET UTF8, with *n* being the value of this CQD.
Values	The default value is 32000.

Managing Histograms

This section describes these CQDs that are used to manage histograms:

- "CACHE_HISTOGRAMS_REFRESH_INTERVAL" (page 466)
- "HIST_NO_STATS_REFRESH_INTERVAL" (page 467)
- "HIST_PREFETCH" (page 467)
- "HIST_ROWCOUNT_REQUIRING_STATS" (page 468)

CACHE_HISTOGRAMS_REFRESH_INTERVAL

Category	Histograms
Description	Defines the time interval after which timestamps for cached histograms are checked to be refreshed.
Values	Unsigned integer Unit is seconds. The default value is '3600' (1 hour).
Usage	Histogram statistics are cached so that the compiler can avoid access to the metadata tables, thereby reducing compile times. The timestamp of the tables are checked against those of the cached histograms at an interval specified by this CQD, in order to see if the cached histograms need to be refreshed. You can increase the interval to reduce the impact on compile times as long as you do not need to obtain fresh statistics more frequently in order to improve query performance. It may

	be that the default interval is too long and you would rather refresh the statistics more frequently than the default one hour, in order to improve query performance at the cost of increased compile times. This setting depends on how frequently you are updating statistics on tables. There is no point in refreshing statistics frequently when statistics are not being updated during that time. On the other hand if you are updating statistics, or generating them for the first time on freshly loaded tables frequently enough, and you want these to be picked up immediately by the compiler because you have seen this to have a dramatic impact on plan quality, then you can make the refresh more frequent.
Production usage	Not applicable
Impact	Longer histogram refresh intervals can improve compile times. However, the longer the refresh interval the more obsolete the histograms. That could result in poor performance for queries that could leverage recently updated statistics.
Level	System or Service
Addressing the real problem	Not applicable

HIST_NO_STATS_REFRESH_INTERVAL

Category	Histograms
Description	Defines the time interval after which the fake histograms in the cache should be refreshed unconditionally.
Values	Integer — Unit is seconds. The default value is '3600' (1 hour).
Usage	Histogram statistics are "fake" when update statistics is not being run, but instead the customer is updating the histogram tables directly with statistics to guide the optimizer. This may be done if the data in the table is very volatile (such as for temporary tables), update statistics is not possible because of constant flush and fill of the table occurring, and statistics are manually set to provide some guidance to the optimizer to generate a good plan. If these fake statistics are updated constantly to reflect the data churn, this default can be set to 0. This would ensure that the histograms with fake statistics are not cached, and are always refreshed. If these fake statistics are set and not touched again, then this interval could be set very high.
Production usage	Not applicable
Impact	Setting a high interval improves compilation time. However, if statistics are being updated, the compiler may be working with obsolete histogram statistics, potentially resulting in poorer plans.
Level	Service
Conflicts/Synergies	Not applicable
Addressing the real problem	Not applicable

HIST_PREFETCH

Category	Histograms
Description	Influences the compiler to pre-fetch the histograms and save them in cache.
Values	'ON' — Pre-fetches the histograms. 'OFF' — Does not pre-fetch the histograms. The default value is 'ON'.
Usage	You may want to turn this off if you don't want to pre-fetch a large number of histograms, many of which may not be used.

Production usage	Not applicable
Impact	Though it makes compilation time faster, it may result in the histogram cache to be filled with histograms that may never be used.
Level	System or Service
Conflicts/Synergies	Use this CQD with CACHE_HISTOGRAMS. If CACHE_HISTOGRAMS is OFF, then this CQD has no effect.
Addressing the real problem	Not applicable

HIST_ROWCOUNT_REQUIRING_STATS

Category	Histograms
Description	Specifies the minimum row count for which the optimizer needs histograms, in order to compute better cardinality estimates. The optimizer does not issue any missing statistics warnings for tables whose size is smaller than the value of this CQD.
Values	Integer The default value is '50000'.
Usage	Use this CQD to reduce the number of statistics warnings.
Production usage	Not applicable
Impact	Missing statistics warnings are not displayed for smaller tables, which in most cases don't impact plan quality much. However, there may be some exceptions where missing statistics on small tables could result in less than optimal plans.
Level	System
Conflicts/Synergies	Use this CQD with HIST_MISSING_STATS_WARNING_LEVEL. If the warning level CQD is 0, then this CQD does not have any effect. Also, for tables having fewer rows than set in this CQD, no warnings are displayed irrespective of the warning level.
Addressing the real problem	Not applicable

Optimizer

This section describes these CQDs that are used by the Optimizer:

- "JOIN_ORDER_BY_USER" (page 468)
- "MDAM_SCAN_METHOD" (page 469)
- "SUBQUERY_UNNESTING" (page 469)

JOIN_ORDER_BY_USER

Category	Influencing Query Plans
Description	Enables or disables the join order in which the optimizer joins the tables to be the sequence of the tables in the FROM clause of the query.
Values	'ON' Join order is forced. 'OFF' Join order is decided by the optimizer. The default value is 'OFF'.
Usage	When set to ON, the optimizer considers only execution plans that have the join order matching the sequence of the tables in the FROM clause.
Production usage	This setting is to be used only for forcing a desired join order that was not generated by default by the optimizer. It can be used as a workaround for query plans with inefficient join order.

Impact	Because you are in effect forcing the optimizer to use a plan that joins the table in the order specified in the FROM clause, the plan generated may not be the optimal one.
Level	Query
Conflicts/Synergies	Not applicable
Addressing the real problem	Not applicable

MDAM_SCAN_METHOD

Category	Influencing Query Plans
Description	Enables or disables the Multi-Dimensional Access Method.
Values	'ON' MDAM is considered. 'OFF' MDAM is disabled. The default value is 'ON'.
Usage	In certain situations, the optimizer might choose MDAM inappropriately, causing poor performance. In such situations you may want to turn MDAM OFF for the query it is effecting.
Production usage	Not applicable
Impact	Table scans with predicates on non-leading clustering key column(s) could benefit from MDAM access method if the leading column(s) has a small number of distinct values. Turning MDAM off results in a longer scan time for such queries.
Level	Set this CQD at the query level when MDAM is not working efficiently for a specific query. However, there may be cases (usually a defect) where a larger set of queries is being negatively impacted by MDAM. In those cases you may want to set it at the service or system level.
Conflicts/Synergies	Not applicable
Addressing the real problem	Not applicable

SUBQUERY_UNNESTING

Category	Influencing Query Plans
Description	Controls the optimizer's ability to transform nested sub-queries into regular join trees.
Values	'ON' Subquery un-nesting is considered. 'OFF' Subquery un-nesting is disabled. The default value is 'ON'.
Usage	Use this control to disable subquery un-nesting in the rare situation when un-nesting results in an inefficient query execution plan
Production usage	Not applicable
Impact	In general, subquery un-nesting results in more efficient execution plans for queries with nested sub-queries. Use only as a workaround for observed problems due to un-nesting.
Level	Query
Conflicts/Synergies	Not applicable
Addressing the real problem	Not applicable

Managing Schemas

This section describes these CQDs that are used for managing schemas:

- "SCHEMA" (page 470)

SCHEMA

Category	Schema controls
Description	Sets the default schema for the session.
Values	SQL identifier The default is SEABASE.
Usage	A SET SCHEMA statement, or a CONTROL QUERY DEFAULT SCHEMA statement, can be used to override the default schema name.
Production usage	It is a convenience so you do not have to type in two-part names.
Impact	Not applicable
Level	Any
Conflicts/Synergies	Alternately you can use the SET SCHEMA statement.
Addressing the real problem	Not applicable

Transaction Control and Locking

This section describes these CQDs that are used for transaction control and locking:

- "BLOCK_TO_PREVENT_HALLOWEEN" (page 470)
- "UPD_ORDERED" (page 471)

BLOCK_TO_PREVENT_HALLOWEEN

Category	Runtime controls
Description	A self-referencing insert is one which inserts into a target table and also scans from the same target table as part of the query that produces rows to be inserted. Inconsistent results are produced by the insert statement if the statement scans rows which have been inserted by the same statement. This is sometimes called the "Halloween problem." Trafodion prevents the Halloween problem using one of two methods: 1) the blocking method uses a SORT operation to ensure all rows have been scanned before any are inserted, or 2) the disk process (ESAM) locks method tracks the rows which have already been inserted and the SCAN operator skips these rows. The compiler chooses the blocking method in cases in which static analysis of the plan indicates that the disk process locks method cannot be used. However, the compiler does not evaluate one condition that would prevent the use of the disk process locks method: the AUTOCOMMIT setting in which the statement is executed. Instead the compiler assumes that the statement is executed with the default setting for AUTOCOMMIT, 'ON'. If AUTOCOMMIT is set to 'OFF' and self-referencing insert statement which uses the disk process locks method is executed, then a runtime error (SQLCODE 8107) is raised. This CQD is used to force the compiler to use the blocking method to prevent error 8107.
Values	'OFF' — The compiler is free to choose which method to use to prevent the Halloween problem. 'ON' — The compiler is forced to use the blocking method. The default value is 'ON'.
Usage	Change this default to 'ON' if error 8107 is raised for a self-referencing insert statement which is executed in a session with AUTOCOMMIT set to 'OFF'.
Production usage	Not applicable

Impact	Using the 'ON' value in conditions that require it allows successful completion of the insert statement. Using the 'ON' value when not required can decrease performance of some self-referencing insert statements.
Level	If self-referencing insert statements which execute with AUTOCOMMIT 'OFF' can be restricted to a service level, then this default should be set to 'ON' only for that service level. Otherwise the setting should be made for the system.
Conflicts/Synergies	Not applicable
Addressing the real problem	Not applicable

UPD_ORDERED

Category	Influencing Query Plans
Description	Controls whether rows should be inserted, updated, or deleted in clustering key order.
Values	'ON' — The optimizer generates and considers plans where the rows are inserted, updated, or deleted in clustering key order. 'OFF' — The optimizer does not generate plans where the rows must be inserted, updated, or deleted in clustering key order. The default value is 'ON'.
Usage	Inserting, updating or deleting rows in the clustering key order is most efficient and highly recommended. Turning this CQD OFF may result in saving the data sorting cost but at the expense of having less efficient random I/O Insert/Update/Delete operations. If you know that he data is already sorted in clustering key order, or is mostly in clustering key order, so that it would not result in random I/O, you could set this CQD to OFF.
Production usage	Not applicable
Impact	If turned OFF, the system may perform large number of inefficient Random I/Os when performing Insert/Update/Delete operations.
Level	Query
Conflicts/Synergies	Not applicable
Addressing the real problem	Not applicable

C Limits

This appendix lists limits for various parts of Trafodion SQL.

Column names	Up to 128 characters long, or 256 bytes of UTF8 text, whichever is less.
Schema names	Up to 128 characters long, or 256 bytes of UTF8 text, whichever is less.
Table names	ANSI names are of the form *schema*.*object*, where each part can be up to 128 characters long, or 256 bytes of UTF8 text, whichever is less.

Index

A

ABS function
  examples of, 285
  syntax diagram of, 285
Access options
  summary of, 25
  DELETE statement use of, 86
  DML statements use of, 25
  INSERT statement use of, 118
  READ COMMITTED, 25
  SELECT statement use of, 146
  UPDATE statement use of, 170
Access privileges
  tables, 111
ACOS function
  examples of, 286
  syntax diagram of, 286
Active query, 443
ADD_MONTHS function
  examples of, 287
  syntax diagram of, 287
Aggregate functions
  summary of, 278
  AVG, 293
  COUNT, 313
  DISTINCT clause, 147, 278
  MAX/MAXIMUM, 360
  MIN, 361
  STDDEV, 410
  SUM, 414
  VARIANCE, 426
Aliases, schemas, 249
ALTER LIBRARY statement, 34
  considerations for, 34
  examples of, 34
  syntax diagram of, 34
ALTER TABLE statement
  authorization and availability requirements, 40
  considerations for, 40
  constraints implemented with indexes, 40
  DEFAULT clause, 257
  description for, 37
  examples of, 40
  RENAME TO clause, 39
  syntax diagram of, 36
ALTER USER statement
  considerations for, 41
  examples of, 41
  syntax diagram of, 41
ANSI
  compliance, description of, 27
  names, schemas, 249
ASCII function
  examples of, 288
  syntax diagram of, 288
ASIN function
  examples of, 289
  syntax diagram of, 289
ATAN function
  examples of, 290
  syntax diagram of, 290
ATAN2 function
  examples of, 291
  syntax diagram of, 291
AUTHNAME function
  considerations for, 292
  example of, 292
  syntax diagram of, 292
Authorization ID
  description of, 193
Automated UPDATE STATISTICS, 190
AVG function
  DISTINCT clause use of, 293
  examples of, 293
  operand requirements, 293
  syntax diagram of, 293
AVG window function
  examples of, 433
  syntax diagram of, 433

B

BEGIN WORK statement
  considerations for, 42
  example of, 42
  syntax diagram of, 42
BETWEEN predicate
  examples of, 234
  logical equivalents, 234
  operand requirements, 233
  syntax diagram of, 233
Big memory operators (BMOs), 442
Bignum, 59
BITAND function
  examples of, 295
  syntax diagram of, 295
BLOCK_TO_PREVENT_HALLOWEEN, 470
Boolean operators
  NOT, AND, OR, 250
  search condition use of, 250
Bulk Loader, 177
Bulk Unloader, 183

C

CACHE_HISTOGRAMS_REFRESH_INTERVAL, 466
CALL statement
  considerations for, 43
  description for, 43
  examples of, 44
  required privileges, 43
  syntax diagram of, 43
  usage restrictions, 43

CASE expression
data type of, 297
examples of, 297
searched CASE form, 296
syntax diagram of, 296
CAST expression
data type conversion, 299
examples of, 300
syntax diagram of, 299
valid type combinations, 299
CEILING function
examples of, 301
syntax diagram of, 301
CHAR data type, 204, 205
CHAR function
examples of, 302
syntax diagram of, 302
CHAR VARYING data type, 204
CHAR_LENGTH function
examples of, 303
syntax diagram of, 303
Character sets
setting default, 224
supported in Trafodion SQL, 193
Character string data types
CHAR and VARCHAR, differences, 205
Character string functions
summary of, 279
ASCII, 288
CHAR, 302
CHAR_LENGTH , 303
CODE_VALUE, 305
CONCAT, 306
INSERT, 348
LCASE, 352
LEFT, 353
LOCATE, 354
LOWER, 357
LPAD, 358
LTRIM, 359
OCTET_LENGTH, 378
POSITION, 381
REPEAT, 388
REPLACE, 389
RIGHT, 390
RPAD, 395
RTRIM, 396
SPACE, 408
SUBSTRING/SUBSTR, 412
TRANSLATE, 420
TRIM, 421
UCASE, 422
UPPER, 423
UPSHIFT, 424
Character string literals, 224
Character value expression
examples of, 212
syntax diagram of, 211
CHECK constraint, 38, 39, 73, 195
Clauses
DEFAULT, 257
FORMAT, 259
SAMPLE, 261
SEQUENCE BY, 268
TRANSPOSE, 271
Clustering key
description of, 223
COALESCE function
example of, 304
syntax diagram of, 304
CODE_VALUE function
example of, 305
syntax diagram of, 305
Columns
column reference, 193
default values, 194
description of, 193
qualified name, 193
COMMIT WORK statement
considerations for, 46
example of, 46
syntax diagram of, 46
Comparable data types, 119, 201
Comparison predicates
comparison operators, 235
data conversions, 236
examples of, 236
operand requirements, 235
syntax diagram of, 235
Compatible data types, 119, 201
Component privilege statements
GRANT COMPONENT PRIVILEGE, 114
REVOKE COMPONENT PRIVILEGE, 133
CONCAT function
examples of, 306
syntax diagram of, 306
Concatenation operator (||)
description of, 306
examples of, 303, 306
Concurrency
DELETE statement, 87
description of, 26
INSERT statement, 119
UPDATE statement, 170
Constraints
CHECK, 38, 39, 73, 195
description of, 195
FOREIGN KEY, 38, 39, 73, 195
NOT NULL, 38, 72, 195
PRIMARY KEY, 72, 195
UNIQUE, 38, 39, 72, 195
CONTROL QUERY CANCEL statement
considerations for, 47
example of, 48
syntax diagram of, 47
CONTROL QUERY DEFAULT statement
considerations for, 49
examples of, 49

syntax diagram of, 49
CONVERTTIMESTAMP function
examples of, 310
JULIANTIMESTAMP inverse relationship to, 310
syntax diagram of, 310
CONVERTTOHEX function
examples of, 308
syntax diagram of, 308
Correlation names
examples of, 196
purpose of, 196
table reference use of, 196
COS function
examples of, 311
syntax diagram of, 311
COSH function
examples of, 312
syntax diagram of, 312
COUNT function
DISTINCT clause within, 313
examples of, 313
syntax diagram of, 313
COUNT window function
examples of, 434
syntax diagram of, 434
Counters, 447
CREATE FUNCTION statement, 50
considerations for, 52
examples of, 52
syntax diagram of, 50
CREATE INDEX statement
authorization requirements , 55
considerations for, 54
examples of, 55
limits on indexes, 55
syntax diagram of, 53
volatile index, 53
CREATE LIBRARY statement, 56
considerations for, 56
examples of, 57
syntax diagram of, 56
CREATE PROCEDURE statement, 58
considerations for, 62
examples of, 63
syntax diagram of, 58
CREATE ROLE statement
considerations for, 66
examples of, 66
syntax diagram of, 66
CREATE SCHEMA statement, 249
examples of, 68
reserved schema names, 67
syntax diagram of, 67
CREATE TABLE AS statement
examples of, 79
LOAD IF EXISTS, 78
syntax diagram of, 69
CREATE TABLE LIKE statement
COMPRESSION attribute, 77
considerations for, 77
file attributes, 77
syntax diagram of, 69
CREATE TABLE statement
considerations for volatile tables, 75
DEFAULT clause, 257
examples of, 79
STORE BY clause, 70
syntax diagram of, 69
Trafodion SQL extensions, 78
CREATE VIEW statement
considerations for, 82
examples of, 84
ORDER BY clause, 83
syntax diagram of, 81
updatability requirements, 83
vertical partition example, 84
WITH CHECK OPTION within, 81
CREATE VOLATILE INDEX statement
syntax diagram of, 53
CREATE VOLATILE TABLE statement
considerations for, 75
examples of, 77
nullable constraints, 76
nullable keys, 75
restrictions, 75
suitable keys, 75
syntax diagram of, 69
CROSS JOIN, description of, 143
CURRENT_DATE function
examples of, 316
syntax diagram of, 316
CURRENT_TIME function
examples of, 317
precision specification within, 317
syntax diagram of, 317
CURRENT_TIMESTAMP function
examples of, 315, 318
precision specification within, 315, 318
syntax diagram of, 315, 318
CURRENT_USER function
considerations for, 319
example of, 319
syntax diagram of, 319

D

Data Definition Language (DDL) statements
summary of, 30
ALTER LIBRARY, 34
ALTER TABLE, 36
ALTER USER, 41
CREATE FUNCTION, 50
CREATE INDEX , 53
CREATE LIBRARY, 56
CREATE PROCEDURE, 58
CREATE ROLE, 66
CREATE SCHEMA, 67
CREATE TABLE, 69
CREATE VIEW, 81

DROP FUNCTION, 88
DROP INDEX, 89
DROP LIBRARY, 90
DROP PROCEDURE, 92
DROP ROLE, 93
DROP SCHEMA, 95
DROP TABLE, 96
DROP VIEW, 97
GRANT, 111
GRANT COMPONENT PRIVILEGE, 114
GRANT ROLE, 117
REGISTER USER, 128
REVOKE, 130
REVOKE COMPONENT PRIVILEGE, 133
REVOKE ROLE, 135
UNREGISTER USER, 168
Data Manipulation Language (DML) statements
summary of, 31
DELETE, 86
INSERT, 118
MERGE, 123
SELECT, 138
TABLE, 167
UPDATE, 169
VALUES, 175
Data type conversion, CAST expression, 299
Data types
approximate numeric
descriptions of, 209
DOUBLE PRECISION, 210
FLOAT, 209
REAL, 210
character, 204
comparable and compatible, 201
datetime
DATE, 205
TIME, 205
TIMESTAMP, 205
exact numeric
DECIMAL, 209
descriptions of, 209
INTEGER, 209
LARGEINT, 209
NUMERIC, 209
SMALLINT, 209
extended numeric precision, 202
fixed length character
CHAR, 204
NATIONAL CHAR, 204
NCHAR, 204
interval, 207
Java, 60
literals, examples of
datetime literals, 226
interval literals, 229
numeric literals, 230
SQL, 60
varying-length character
CHAR VARYING, 204
NATIONAL CHAR VARYING, 204
NCHAR VARYING, 204
VARCHAR, 204
Database object name, 198
Database objects, 197
DATE_ADD function
examples of, 320
syntax diagram of, 320
DATE_PART (of a timestamp) function
examples of, 326
syntax diagram of, 326
DATE_PART (of an interval) function
examples of, 325
syntax diagram of, 325
DATE_SUB function
examples of, 321
syntax diagram of, 321
DATE_TRUNC function
examples of, 327
syntax diagram of, 327
DATEADD function
examples of, 322
syntax diagram of, 322
DATEDIFF function
examples of, 323
syntax diagram of, 323
DATEFORMAT function
examples of, 324
syntax diagram of, 324
Datetime data types
DATE, 205
description of, 205
examples of literals, 226
TIME, 205
TIMESTAMP, 205
Datetime functions
summary of, 280
ADD_MONTHS, 287
CONVERTTIMESTAMP, 310
CURRENT, 315
CURRENT_DATE, 316
CURRENT_TIME, 317
CURRENT_TIMESTAMP, 318
DATE_ADD, 320
DATE_PART (interval), 325
DATE_PART (timestamp), 326
DATE_SUB, 321
DATE_TRUNC, 327
DATEADD, 322
DATEDIFF, 323
DATEFORMAT, 324
DAY, 328
DAYNAME, 329
DAYOFMONTH, 330
DAYOFWEEK, 331
DAYOFYEAR, 332
EXTRACT, 345
HOUR, 347
JULIANTIMESTAMP, 350

MINUTE, 362
MONTH, 364
MONTHNAME, 365
QUARTER, 383
SECOND, 404
TIMESTAMPADD, 418
TIMESTAMPDIFF, 419
WEEK, 428
YEAR, 429
Datetime literals
description of, 226
Datetime value expression
examples of, 213
syntax diagram of, 212
DAY function
examples of, 328
syntax diagram of, 328
DAYNAME function
examples of, 329
syntax diagram of, 329
DAYOFMONTH function
examples of, 330
syntax diagram of, 330
DAYOFWEEK function
examples of, 331
syntax diagram of, 331
DAYOFYEAR function
examples of, 332
syntax diagram of, 332
DDL statements *see* Data Definition Language (DDL) statements
DECIMAL data type, 209
DECODE function
examples of, 334
syntax diagram of, 333
DEFAULT clause
ALTER TABLE use of, 37
CREATE TABLE use of, 72, 257
examples of, 257
syntax diagram of, 257
Default settings
changing, 49
DEGREES function
examples of, 336
syntax diagram of, 336
DELETE statement
access options, 86
authorization requirements, 86
considerations for, 86
examples of, 87
isolation levels, 87
syntax diagram of, 86
WHERE clause, 86
Delimited identifiers, 221
DENSE_RANK window function
examples of, 435
syntax diagram of, 435
Derived column names
examples of, 194
syntax of, 193
DETAIL_COST in EXPLAIN output
CPU_TIME, 344
IDLETIME, 344
IO_TIME, 344
MSG_TIME, 344
PROBES, 344
DIFF1 function
equivalent definitions, 337
examples of, 337
syntax diagram of, 337
DIFF2 function
equivalent definitions, 339
examples of, 339
syntax diagram of, 339
DISTINCT clause
aggregate functions, 147, 278
AVG function use of, 293
COUNT function use of, 313
MAX function use of, 360
MAXIMUM function use of, 360
MIN function use of, 361
STDDEV function use of, 410
SUM function use of, 414
VARIANCE function use of, 426
DML statements *see* Data Manipulation Language (DML) statements
DOUBLE PRECISION data type, 210
DROP FUNCTION statement, 88
considerations for, 88
examples of, 88
syntax diagram of, 88
DROP INDEX statement
authorization and availability requirements, 89
considerations for, 89
example of, 89
syntax diagram of, 89
DROP LIBRARY statement, 90
considerations for, 90
examples of, 90
syntax diagram of, 90
DROP PROCEDURE statement, 92
considerations for, 92
examples of, 92
syntax diagram of, 92
DROP ROLE statement
considerations for, 93
examples of, 93
syntax diagram of, 93
DROP SCHEMA statement, 249
authorization and availability requirements, 95
considerations for, 95
example of, 95
syntax diagram of, 95
DROP TABLE statement
authorization requirements, 96
considerations for, 96
examples of, 96
syntax diagram of, 96

DROP VIEW statement
authorization requirements, 97
considerations for, 97
example of, 97
syntax diagram of, 97
DROP VOLATILE INDEX statement
considerations for, 89
examples of, 89
syntax diagram of, 89
DROP VOLATILE TABLE statement
authorization requirements, 96
examples of, 96
syntax diagram of, 96

E

Error messages, 29
EXECUTE statement
considerations for, 99
examples of, 99
scope of, 99
syntax diagram of, 98
EXISTS predicate
correlated subquery within, 238
examples of, 238
syntax diagram of, 238
EXP function
examples of, 341
syntax diagram of, 341
EXPLAIN function
columns in result, 343
examples of, 344
operator tree, 343
plan in running query, 342
syntax diagram of, 342
Explain statement
displayed, 101
examples of, 102
operators, 101
OPTIONS 'f' considerations, 102
plan in running query, 101
reviewing query execution plans, 101
syntax, 101
Expression
character (or string) value, 211
datetime value, 212, 213, 217
description of, 211
interval value, 213, 217
numeric value, 218
Extended numeric precision, 202
Extensions, statements, 28
EXTRACT function
examples of, 345
syntax diagram of, 345

F

File options
CREATE TABLE use of, 69
Fixed-length character column, 205
FLOAT data type, 209
FLOOR function
examples of, 346
syntax diagram of, 346
FOREIGN KEY constraint, 39, 73, 195
FORMAT clause
considerations for date formats, 260
considerations for other formats, 260
examples of, 260
syntax diagram, 259
FULL join, description of, 143
Function statements, 32
Functions, ANSI compliant, 28

G

GET HBASE OBJECTS statement
examples of, 107
syntax diagram of, 107
GET statement
considerations for, 104
examples of, 105
syntax diagram of, 103
GET STATISTICS command
process ID (PID), 443
query ID (QID), 443
syntax, 443
GET VERSION OF METADATA statement
considerations for, 109
examples of, 109
syntax diagram of, 109
GET VERSION OF SOFTWARE statement
considerations for, 110
examples of, 110
syntax diagram of, 110
GRANT COMPONENT PRIVILEGE statement, 114
authorization and availability requirements, 116
considerations for, 116
example of, 116
syntax diagram of, 114
GRANT ROLE statement
considerations for, 117
example of, 117
syntax diagram of, 117
GRANT statement
authorization and availability requirements, 112
considerations for, 112
examples of, 112
syntax diagram of, 111

H

HBASE_INTERFACE, 466
HIST_NO_STATS_REFRESH_INTERVAL, 467
HIST_PREFETCH, 467
HIST_ROWCOUNT_REQUIRING_STATS, 468
Histograms
clearing, 186
UPDATE STATISTICS use of, 187
HIVE_MAX_STRING_LENGTH, 466
HOUR function
examples of, 347

syntax diagram of, 347

I
Identifiers, 221
IN predicate
examples of, 240
logical equivalent using ANY, 240
operand requirements, 240
syntax diagram of, 239
Index keys, 223
Indexes
and transactions, 54
CREATE INDEX statement, 53
CREATE VOLATILE INDEX statement, 53
description of, 222
DROP INDEX statement, 89
LOAD statement, 177
UNLOAD statement, 183
INSERT function
examples of, 348
syntax diagram of, 348
INSERT statement
compatible data types, 119
considerations for, 118
examples of, 120
self-referencing, 119
side tree, 55
syntax diagram of, 118
target column list, 118
VALUES specification within, 119
INTEGER data type, 209
Interval data types
description of, 207
Interval literals
description of, 227
examples of, 229
Interval value expression
examples of, 216
syntax diagram of, 215
INVOKE statement
considerations for, 122
example of, 122
syntax diagram of, 122
ISNULL function
examples of, 349
syntax diagram of, 349
Isolation levels
READ COMMITTED, 26

J
Java data types, 60
Join
CROSS, 143
FULL, 143
JOIN ON, 143
join predicate, 151
LEFT, 143
limits, 147
NATURAL, 143
NATURAL FULL, 143
NATURAL LEFT, 143
NATURAL RIGHT, 143
optional specifications, 142
RIGHT, 143
types, 142
JOIN ON join, description of, 143
JOIN_ORDER_BY_USER, 468
JULIANTIMESTAMP function
examples of, 350
syntax diagram of, 350

K
Keys
clustering, 223
index, 223
primary, 223
SYSKEY, 223

L
LARGEINT data type, 209
LASTNOTNULL function
examples of, 351
syntax diagram of, 351
LCASE function
examples of, 352
syntax diagram of, 352
LEFT function
examples of, 353
syntax diagram of, 353
LEFT join, description of, 143
LIKE predicate
considerations for, 241
syntax of, 241
Limits
IN predicate, 239
indexes, 55
number of tables joined, 147
tables, 472
Literals
datetime, examples of, 226
description of, 224
examples of, 224
interval, examples of, 229
numeric, examples of, 230
LOAD statement
considerations for, 178
example of, 179
syntax diagram of, 177
Loading data into tables
LOAD IF EXISTS option, 78
LOCATE function
examples of, 354
result of, 354
syntax diagram of, 354
Locking
READ COMMITTED access option, 25
LOG function
examples of, 355

syntax diagram of, 355
LOG10 function
examples of, 356
syntax diagram of, 356
Logical name
Trafodion SQL objects, 198
Logical operators
NOT, AND, OR, 250
search condition use of, 250
LOWER function
examples of, 357
syntax diagram of, 357
LPAD function
examples of, 358
syntax diagram of, 358
LTRIM function
examples of, 359
syntax diagram of, 359

M

Magnitude, 220
Math functions
summary of, 281
ABS, 285
ACOS, 286
ASIN, 289
ATAN, 290
ATAN2, 291
CEILING, 301
COS, 311
COSH, 312
DEGREES, 336
EXP, 341
FLOOR, 346
LOG, 355
LOG10, 356
MOD, 363
NULLIFZERO, 376
PI, 380
POWER, 382
RADIANS, 384
ROUND, 391
SIGN, 405
SIN, 406
SINH, 407
SQRT, 409
TAN, 415
TANH, 416
ZEROIFNULL, 430
MAX function
considerations for, 360
DISTINCT clause within, 360
examples of, 360
syntax diagram of, 360
MAX window function
examples of, 436
syntax diagram of, 435
MAXIMUM function
considerations for, 360
DISTINCT clause within, 360
examples of, 360
syntax diagram of, 360
MDAM_SCAN_METHOD, 469
MERGE statement
considerations for, 123
description of, 123
example of, 125
merge from one table to another, 125
restrictions, 124
syntax diagram of, 123
upsert, single row, 123
MIN function
DISTINCT clause within, 361
examples of, 361
syntax diagram of, 361
MIN window function
examples of, 437
syntax diagram of, 436
MINUTE function
examples of, 362
syntax diagram of, 362
MOD function
examples of, 363
syntax diagram of, 363
MONTH function
examples of, 364
syntax diagram of, 364
MONTHNAME function
examples of, 365
syntax diagram of, 365
MOVINGAVG function
examples of, 366
syntax diagram of, 366
MOVINGCOUNT function
examples of, 367
syntax diagram of, 367
MOVINGMAX function
examples of, 368
syntax diagram of, 368
MOVINGMIN function
examples of, 369
syntax diagram of, 369
MOVINGSTDDEV function
examples of, 370
syntax diagram of, 370
MOVINGSUM function
examples of, 372
syntax diagram of, 372
MOVINGVARIANCE function
examples of, 373
syntax diagram of, 373

N

NATIONAL CHAR data type, 204
NATIONAL CHAR VARYING data type, 204
NATURAL FULL join, description of, 143
NATURAL join, description of, 143
NATURAL LEFT join, description of, 143

NATURAL RIGHT join, description of, 143
NCHAR data type, 204
NCHAR VARYING data type, 204
NOT CASESPECIFIC, 72
NOT NULL constraint, 38, 72, 195
NULL predicate
  examples of, 243
  syntax diagram of, 243
Null symbol, 231
NULL, using, 169
NULLIF function
  example of, 375
  syntax diagram of, 375
NULLIFZERO function
  examples of, 376
  syntax diagram of, 376
Numeric data types
  approximate numeric, 209
  exact numeric, 209
  extended numeric , 202
  literals, examples of, 230
Numeric literals
  approximate, 229
  exact, 229
  examples of, 230
Numeric value expression
  evaluation order, 219
  examples of, 220
  syntax diagram of, 218
NVL function
  examples of, 377
  syntax diagram of, 377

O

Object names, 198
Object namespace, 198
Objects
  description of, 198
  logical names, 198
OCTET_LENGTH function
  CHAR_LENGTH similarity to, 378
  examples of, 378
  syntax diagram of, 378
OFFSET function
  examples of, 379
  syntax diagram of, 379
OLAP window functions, 431
  AVG window function, 433
  COUNT window function, 434
  DENSE_RANK window function, 435
  limitations, 432
  MAX window function, 435
  MIN window function, 436
  ORDER BY clause, 431
  RANK window function, 437
  ROW_NUMBER window function, 438
  STDDEV window function, 438
  SUM window function, 439
  VARIANCE window function, 440
Operator statistics, 442
Operators in query execution plan, 101
OPTIONS on Explain statement
  'f', 102
ORDER BY clause
  AS and ORDER BY conflicts, 147
  guidelines for CREATE VIEW, 83
Other functions and expressions
  AUTHNAME, 292
  BITAND , 295
  CASE expression, 296
  CAST expression, 299
  COALESCE, 304
  CONVERTTOHEX, 308
  CURRENT_USER, 319
  DECODE, 333
  EXPLAIN, 342
  ISNULL, 349
  NULLIF, 375
  NVL, 377
  USER, 425

P

Parent query ID (QID), 455
Performance
  CAST, 299
  ORDER BY clause, 148
  updating rows, 170
PERTABLE statistics, 442
PI function
  examples of, 380
  syntax diagram of, 380
POPULATE INDEX utility
  considerations for, 180
  examples of, 181
  syntax diagram of, 180
POSITION function
  examples of, 381
  result of, 381
  syntax diagram of, 381
POWER function
  examples of, 382
  syntax diagram of, 382
Precision, description of, 220
Predicates
  summary of, 233
  BETWEEN, 233
  comparison, 235
  description of, 233
  EXISTS, 238
  IN, 239
  LIKE, 241
  NULL, 243
  quantified comparison, 244
PREPARE statement
  availability, 126
  considerations for, 126
  examples of, 126
  syntax diagram of, 126

Prepared statements, 32, 33
PRIMARY KEY constraint, 72, 195
Primary key, description of, 223
Primary role
  described, 248
Privileges, 247
  GRANT COMPONENT PRIVILEGE statement use of, 114
  GRANT ROLE statement use of, 117
  GRANT statement use of, 111
  REVOKE COMPONENT PRIVILEGE statement use of, 133
  REVOKE ROLE statement use of, 135
  REVOKE statement use of, 130
PURGEDATA utility
  considerations for, 182
  example of, 182
  syntax diagram of, 182

Q

Quantified comparison predicates
  ALL, ANY, SOME, 244
  examples of, 245
  operand requirements, 244
  result of, 245
  syntax diagram of, 244
QUARTER function
  examples of, 383
  syntax diagram of, 383
Query execution plan
  displayed, 101
  in running query, 101
  operators, 101
  reviewing, 101
Query expression
  SELECT statement use of, 141
Query ID (QID)
  child QID, 456
  components of, 459
  extracting, 459
  obtaining, 443
  parent QID, 455
Query specification
  SELECT statement use of, 144
  simple table, form of , 144
QUERYID_EXTRACT function, 459

R

RADIANS function
  examples of, 384
  syntax diagram of, 384
RANK window function
  examples of, 438
  syntax diagram of, 437
RANK/RUNNINGRANK function
  example for, 385
  syntax diagram of, 385
READ COMMITTED, 25
REAL data type, 210
Referential integrity
  FOREIGN KEY constraint, 38, 39, 73
REGISTER USER statement
  considerations for, 128
  examples of, 129
  syntax diagram of, 128
RENAME TO clause, 39
Renaming tables, 39
REPEAT function
  examples of, 388
  syntax diagram of, 388
REPLACE function
  examples of, 389
  syntax diagram of, 389
Reserved
  schema names, 67
  words, Trafodion SQL, 462
Resource control statements
  EXECUTE statement, 98
  PREPARE statement, 126
  UPDATE STATISTICS statement, 186
REVOKE COMPONENT PRIVILEGE statement, 133
  authorization and availability requirements, 134
  considerations for, 134
  example of, 134
  syntax diagram of, 133
REVOKE ROLE statement
  considerations for, 135
  examples of, 136
  syntax diagram of, 135
REVOKE statement
  authorization and availability requirements, 131
  considerations for, 131
  examples of, 131
  syntax diagram of, 130
RIGHT function
  examples of, 390
  syntax diagram of, 390
RIGHT join, description of, 143
RMS
  adaptive statistics, 442
  big memory operators, 442
  counters, 447
  displaying, 443
  features, 442
  operator statistics, 442
  overview, 442
  PERTABLE statistics, 442
  statistics about RMS, 457
Roles
  CREATE ROLE statement use of, 66
  description of, 248
  DROP ROLE statement use of, 93
  GRANT ROLE statement use of, 117
  REVOKE ROLE statement use of, 135
ROLLBACK WORK statement
  considerations for, 137
  example of, 137
  syntax diagram of, 137

ROUND function
examples of, 391
syntax diagram of, 391
Row value constructor
BETWEEN predicate use of, 233
comparison predicates use of, 235
IN predicate use of, 239
NULL predicate use of, 243
quantified comparison predicates use of, 244
ROW_NUMBER window function
examples of, 438
syntax diagram of, 438
ROWS SINCE CHANGED function
considerations for, 394
examples of, 394
syntax diagram of, 394
ROWS SINCE function
examples of, 392
syntax diagram of, 392
RPAD function
examples of, 395
syntax diagram of, 395
RTRIM function
examples of, 396
syntax diagram of, 396
RUNNINGAVG function
equivalent definition, 397
examples of, 397
syntax diagram of, 397
RUNNINGCOUNT function
examples of, 398
syntax diagram of, 398
RUNNINGMAX function
examples of, 399
syntax diagram of, 399
RUNNINGMIN function
examples of, 400
syntax diagram of, 400
RUNNINGSTDDEV function
equivalent definition, 401
examples of, 401
syntax diagram of, 401
RUNNINGSUM function
examples of, 402
syntax diagram of, 402
RUNNINGVARIANCE function
examples of, 403
syntax diagram of, 403
Runtime Management System (RMS) *see* RMS
Runtime statistics *see* RMS

S

SAMPLE clause
examples of, 262
SELECT statement use of, 261
syntax diagram of, 261
Scale, 220
SCHEMA, 470
Schemas
creating, 249
description of, 249
dropping, 249
reserved, 67
Search condition
Boolean operators within, 250
CASE expression use of, 297
DELETE statement use of, 86
description of, 252
examples of, 251
predicate within, 250
syntax diagram of, 250
UPDATE statement use of, 170
SECOND function
examples of, 404
syntax diagram of, 404
SELECT statement
access options, 146
authorization requirements, 146
considerations for, 146
DISTINCT clause , 140
embedded delete, 141
embedded insert, 142
embedded update, 142
examples of, 150
FROM clause , 141
GROUP BY clause , 145, 148
HAVING clause, 145
joined table within, 142
LIMIT clause, 146
limit on join tables, 147
ORDER BY clause , 146, 148
RETURN list, 141
select list elements, 140
SEQUENCE BY clause, 145
simple table within, 143
syntax diagram of, 138
TRANSPOSE clause, 144
UNION ALL operation, 150
union operation within, 146, 149
views and, 146
WHERE clause, 144
Self-referencing INSERT, 119
SEQUENCE BY clause
examples of, 269
SELECT statement use of, 268
syntax diagram of, 268
Sequence functions
summary of, 282
DIFF1, 337
DIFF2, 339
LASTNOTNULL, 351
MOVINGAVG, 366
MOVINGCOUNT, 367
MOVINGMAX, 368
MOVINGMIN, 369
MOVINGSTDDEV, 370
MOVINGSUM, 372
MOVINGVARIANCE, 373

OFFSET, 379
RANK/RUNNINGRANK, 385
ROWS SINCE, 392
ROWS SINCE CHANGED, 394
RUNNINGAVG, 397
RUNNINGCOUNT, 398
RUNNINGMAX, 399
RUNNINGMIN, 400
RUNNINGSTDDEV, 401
RUNNINGSUM, 402
RUNNINGVARIANCE, 403
THIS, 417
Set functions, 278
SET PARAM command, 98
SET SCHEMA statement
considerations for, 156
example of, 156
syntax diagram of, 156
SET TRANSACTION statement
autocommit option, 157
considerations for, 157
examples of, 157
explicit transactions, 157
implicit transactions, 157
syntax diagram of, 157
SHOWCONTROL statement
example of, 159
syntax diagram of, 159
SHOWDDL SCHEMA statement
considerations for, 163
example of, 163
syntax diagram of, 163
SHOWDDL statement
considerations for, 160
differences between output and original DDL, 161
examples of, 161
for a user or a role, 160
syntax diagram of, 160
SHOWSTATS statement
examples of, 165
syntax diagram of, 164
Side tree insert
definition of, 55
SIGN function
examples of, 405
syntax diagram of, 405
Simple table, in SELECT statement, 143
SIN function
examples of, 406
syntax diagram of, 406
SINH function
examples of, 407
syntax diagram of, 407
SMALLINT data type, 209
SPACE function
examples of, 408
syntax diagram of, 408
SQL data types, 60
SQL runtime statistics *see* RMS
SQL statements
ANSI compliant, 27
Trafodion SQL extensions, 28
SQL Utilities
Automated UPDATE STATISTICS, 190
POPULATE INDEX utility, 180
PURGEDATA utility, 182
SQL value expression, 211
SQRT function
examples of, 409
syntax diagram of, 409
Statements, SQL
ANSI compliant, 27
Trafodion SQL extensions, 28
Statistics, 442
clearing, 186
UPDATE STATISTICS statement, 189, 191
STATISTICS table-valued function, 460
TDB_ID detail, 460
STDDEV function
DISTINCT clause within, 410
examples of, 411
statistical definition of, 410
syntax diagram of, 410
STDDEV window function
examples of, 439
syntax diagram of, 438
STORE BY clause, 70
Stored procedure statements, 32
ALTER LIBRARY, 34
CALL, 43
CREATE LIBRARY, 56
CREATE PROCEDURE, 58
DROP LIBRARY, 90
DROP PROCEDURE, 92
Stored text
reserved words, 462
String literals, 224
String value expression
examples of, 212
syntax diagram of, 211
Subquery
correlated, 238, 252
description of, 252
inner query, 252
outer query, 252
outer reference, 252
row
BETWEEN predicate, 233
comparison predicate, 235
IN predicate, 239
NULL predicate, 243
quantified comparison predicate, 244
UPDATE statement, 170
UPSERT SELECT statement, 173
scalar
BETWEEN predicate , 233
comparison predicate , 235
IN predicate , 239

NULL predicate, 243
quantified comparison predicate, 244
table, 239
SUBQUERY_UNNESTING, 469
SUBSTR function
examples of, 413
operand requirements, 412
syntax diagram of, 412
SUBSTRING function
examples of, 413
operand requirements, 412
syntax diagram of, 412
Suitable keys
guidelines for selecting, 75
SUM function
DISTINCT clause within, 414
examples of, 414
syntax diagram of, 414
SUM window function
examples of, 440
syntax diagram of, 439
SYSKEY
system-defined clustering key, 223

T

Table
creating, 69
description of, 254
dropping, 96
limits, 472
renaming, 39
subquery, 239
Table reference
description of, 141
SELECT statement use of, 141
TABLE statement
considerations for, 167
examples, 167
syntax diagram for, 167
Table value constructor
description of, 143
simple table, form of, 143
TAN function
examples of, 415
syntax diagram of, 415
TANH function
examples of, 416
syntax diagram of, 416
THIS function
examples of, 417
syntax diagram of, 417
TIMESTAMPADD function
description, 418
examples, 418
TIMESTAMPDIFF function
description, 419
examples, 419
Trafodion SQL objects, logical names, 198
Transaction control statements
BEGIN WORK, 42
COMMIT WORK, 46
ROLLBACK WORK, 137
SET TRANSACTION statement, 157
Transaction isolation levels
READ COMMITTED, 26
Transaction management, 26
AUTOCOMMIT, effect of, 26
BEGIN WORK, 42
COMMIT WORK, 46
ROLLBACK WORK, 137
rules for DML statements, 26
SET TRANSACTION , 157
TRANSLATE function, syntax diagram of, 420
TRANSPOSE clause
cardinality of result, 273
degree of result, 272
examples of, 273
SELECT statement use of, 271
syntax diagram of, 271
TRIM function
examples of, 421
syntax diagram of, 421

U

UCASE function
examples of, 422
syntax diagram of, 422
Union operation
associative, UNION ALL, 150
columns, characteristics of, 149
ORDER BY clause restriction, 149
SELECT statement use of, 146
UNIQUE constraint, 38, 39, 72, 195
UNLOAD statement
considerations for, 184
example of, 184
syntax diagram of, 183
UNREGISTER USER statement
considerations for, 168
examples of, 168
syntax diagram of, 168
UPD_ORDERED, 471
Updatable view, requirements for, 83
UPDATE statement
authorization requirements, 170
conflicting updates, 171
description of, 169
examples of, 172
isolation levels, 170
performance, 170
SET clause, 169
syntax diagram of, 169
WHERE clause, 170
UPDATE STATISTICS
histogram statistics, 189
UPDATE STATISTICS statement
automating, 190
column groups, 187

column lists, 187
considerations, 189
examples of, 191
histogram tables, 187
syntax diagram of, 186
UPPER function
examples of, 423
syntax diagram of, 423
Upsert
using single row, 123
UPSERT statement
examples of, 173
syntax diagram of, 173
UPSHIFT function
syntax diagram of, 424
USER function
considerations for, 425
examples of, 425
syntax diagram of, 425
User-defined function (UDF) statements, 32
Utilities
Automated UPDATE STATISTICS, 190
POPULATE INDEX utility, 180
PURGEDATA utility, 182

V

Value expressions, 211
summary of, 284
CASE (Conditional) expression, 296
CAST expression, 299
VALUES statement
considerations for, 175
examples, 175
syntax diagram for, 175
VARCHAR data type, 204, 205
Variable-length character column, 205
VARIANCE function
DISTINCT clause within, 426
examples of, 427
statistical definition of, 426
syntax diagram of, 426
VARIANCE window function
examples of, 441
syntax diagram of, 440
Vertical partition
example, 84
Views
CREATE VIEW statement, 81
description of, 255
DROP VIEW statement, 97
relationship to tables, 254
updatability requirements, 83
Volatile tables
considerations, 75
examples, 77
nullable constraints, 76
nullable keys, 75
nullable primary key, 76
suitable keys, 75

W

WEEK function
example of, 428
syntax diagram of, 428
window functions
AVG , 433
considerations, 431
COUNT , 434
DENSE_RANK , 435
limitations, 432
MAX , 435
MIN , 436
ORDER BY clause, use of, 431
RANK , 437
ROW_NUMBER, 438
STDDEV, 438
SUM, 439
VARIANCE, 440

Y

YEAR function
examples of, 429
syntax diagram of, 429

Z

ZEROIFNULL function
example of, 430
syntax diagram of, 430