Legal Notice

The information contained herein is subject to change without notice. This documentation is distributed on an “AS IS” basis, without warranties or conditions of any kind, either express or implied. Nothing herein should be construed as constituting an additional warranty. HP shall not be liable for technical or editorial errors or omissions contained herein.

NOTICE REGARDING OPEN SOURCE SOFTWARE: Project Trafodion is licensed under the Apache License, Version 2.0 (the “License”); you may not use software from Project Trafodion except in compliance with the License. You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0. Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

Acknowledgements

Microsoft® and Windows® are U.S. registered trademarks of Microsoft Corporation. Java® is a registered trademark of Oracle and/or its affiliates.
# Contents

### About This Document
- Intended Audience ................................................................. 14
- New and Changed Information in This Edition .............................. 14
- Document Organization .............................................................. 14
- Notation Conventions ............................................................... 14
  - General Syntax Notation ......................................................... 14
- Publishing History ...................................................................... 16
- We Encourage Your Comments .................................................. 17

### 1 Introduction
- SQL Language ........................................................................... 18
- Using Trafodion SQL to Access HBase Tables ............................. 18
  - Initializing the Trafodion Metadata .......................................... 18
  - Accessing Trafodion SQL Tables ............................................ 18
  - Trafodion SQL Tables Versus Native HBase Tables ................. 19
  - Supported SQL Statements With HBase Tables ....................... 19
- Data Consistency and Access Options ........................................ 19
- READ COMMITTED ................................................................. 20
- Transaction Management .......................................................... 20
  - User-Defined and System-Defined Transactions ...................... 20
  - Rules for DML Statements ..................................................... 21
  - Effect of AUTOCOMMIT Option ............................................. 21
  - Concurrency .......................................................................... 21
  - Transaction Isolation Levels .................................................. 21
- ANSI Compliance and Trafodion SQL Extensions ......................... 21
  - ANSI-Compliant Statements .................................................. 22
  - Statements That Are Trafodion SQL Extensions ...................... 22
  - ANSI-Compliant Functions .................................................... 22
- Trafodion SQL Error Messages .................................................. 23

### 2 SQL Statements
- Categories ................................................................................. 24
  - Data Definition Language (DDL) Statements ......................... 24
  - Data Manipulation Language (DML) Statements ...................... 24
  - Control Statements .................................................................. 25
  - Object Naming Statements ..................................................... 25
  - Transaction Control Statements ............................................. 25
- ALTER TABLE Statement .......................................................... 26
  - Syntax Description of ALTER TABLE ....................................... 27
  - Considerations for ALTER TABLE ........................................... 30
  - Example of ALTER TABLE ..................................................... 30
- BEGIN WORK Statement .......................................................... 31
  - Considerations for BEGIN WORK ............................................ 31
  - Example of BEGIN WORK ...................................................... 31
- COMMIT WORK Statement ....................................................... 32
  - Considerations for COMMIT WORK ........................................ 32
  - Example of COMMIT WORK .................................................. 32
- CONTROL QUERY DEFAULT Statement .................................... 33
  - Syntax Description of CONTROL QUERY DEFAULT .................. 33
  - Considerations for CONTROL QUERY DEFAULT .................... 33
  - Examples of CONTROL QUERY DEFAULT ............................ 33
- CREATE INDEX Statement ....................................................... 34
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Authorization IDs</td>
<td>109</td>
</tr>
<tr>
<td>Character Sets</td>
<td>110</td>
</tr>
<tr>
<td>Columns</td>
<td>111</td>
</tr>
<tr>
<td>Column References</td>
<td>111</td>
</tr>
<tr>
<td>Derived Column Names</td>
<td>111</td>
</tr>
<tr>
<td>Column Default Settings</td>
<td>112</td>
</tr>
<tr>
<td>Constraints</td>
<td>113</td>
</tr>
<tr>
<td>Creating or Adding Constraints on SQL Tables</td>
<td>113</td>
</tr>
<tr>
<td>Correlation Names</td>
<td>114</td>
</tr>
<tr>
<td>Explicit Correlation Names</td>
<td>114</td>
</tr>
<tr>
<td>Implicit Correlation Names</td>
<td>114</td>
</tr>
<tr>
<td>Examples of Correlation Names</td>
<td>114</td>
</tr>
<tr>
<td>Database Objects</td>
<td>115</td>
</tr>
<tr>
<td>Examples of VALUES</td>
<td>107</td>
</tr>
<tr>
<td>Syntax Description of VALUES</td>
<td>107</td>
</tr>
<tr>
<td>Considerations for VALUES</td>
<td>107</td>
</tr>
<tr>
<td>Example of VALUES</td>
<td>107</td>
</tr>
<tr>
<td>Syntax Description of VALUES</td>
<td>107</td>
</tr>
<tr>
<td>Considerations for VALUES</td>
<td>107</td>
</tr>
<tr>
<td>Example of VALUES</td>
<td>107</td>
</tr>
<tr>
<td>Syntax Description of UPDATE</td>
<td>96</td>
</tr>
<tr>
<td>Considerations for UPDATE</td>
<td>97</td>
</tr>
<tr>
<td>Examples of UPDATE</td>
<td>99</td>
</tr>
<tr>
<td>Syntax Description of UPDATE</td>
<td>96</td>
</tr>
<tr>
<td>Considerations for UPDATE</td>
<td>97</td>
</tr>
<tr>
<td>Example of UPDATE</td>
<td>99</td>
</tr>
<tr>
<td>Syntax Description of UPDATE</td>
<td>96</td>
</tr>
<tr>
<td>Considerations for UPDATE</td>
<td>97</td>
</tr>
<tr>
<td>Example of UPDATE</td>
<td>99</td>
</tr>
<tr>
<td>Syntax Description of UPDATE</td>
<td>96</td>
</tr>
<tr>
<td>Considerations for UPDATE</td>
<td>97</td>
</tr>
<tr>
<td>Example of UPDATE</td>
<td>99</td>
</tr>
<tr>
<td>Syntax Description of UPDATE</td>
<td>96</td>
</tr>
<tr>
<td>Considerations for UPDATE</td>
<td>97</td>
</tr>
<tr>
<td>Example of UPDATE</td>
<td>99</td>
</tr>
<tr>
<td>Syntax Description of UPDATE</td>
<td>96</td>
</tr>
<tr>
<td>Considerations for UPDATE</td>
<td>97</td>
</tr>
<tr>
<td>Example of UPDATE</td>
<td>99</td>
</tr>
<tr>
<td>Syntax Description of UPDATE</td>
<td>96</td>
</tr>
<tr>
<td>Considerations for UPDATE</td>
<td>97</td>
</tr>
<tr>
<td>Example of UPDATE</td>
<td>99</td>
</tr>
<tr>
<td>Syntax Description of UPDATE</td>
<td>96</td>
</tr>
<tr>
<td>Considerations for UPDATE</td>
<td>97</td>
</tr>
<tr>
<td>Example of UPDATE</td>
<td>99</td>
</tr>
<tr>
<td>Syntax Description of UPDATE</td>
<td>96</td>
</tr>
<tr>
<td>Considerations for UPDATE</td>
<td>97</td>
</tr>
<tr>
<td>Example of UPDATE</td>
<td>99</td>
</tr>
<tr>
<td>Syntax Description of UPDATE</td>
<td>96</td>
</tr>
<tr>
<td>Considerations for UPDATE</td>
<td>97</td>
</tr>
<tr>
<td>Example of UPDATE</td>
<td>99</td>
</tr>
<tr>
<td>Syntax Description of UPDATE</td>
<td>96</td>
</tr>
<tr>
<td>Considerations for UPDATE</td>
<td>97</td>
</tr>
<tr>
<td>Example of UPDATE</td>
<td>99</td>
</tr>
<tr>
<td>Syntax Description of UPDATE</td>
<td>96</td>
</tr>
<tr>
<td>Considerations for UPDATE</td>
<td>97</td>
</tr>
<tr>
<td>Example of UPDATE</td>
<td>99</td>
</tr>
<tr>
<td>Syntax Description of UPDATE</td>
<td>96</td>
</tr>
<tr>
<td>Considerations for UPDATE</td>
<td>97</td>
</tr>
<tr>
<td>Example of UPDATE</td>
<td>99</td>
</tr>
<tr>
<td>Syntax Description of UPDATE</td>
<td>96</td>
</tr>
<tr>
<td>Considerations for UPDATE</td>
<td>97</td>
</tr>
<tr>
<td>Example of UPDATE</td>
<td>99</td>
</tr>
<tr>
<td>Syntax Description of UPDATE</td>
<td>96</td>
</tr>
<tr>
<td>Considerations for UPDATE</td>
<td>97</td>
</tr>
<tr>
<td>Example of UPDATE</td>
<td>99</td>
</tr>
<tr>
<td>Syntax Description of UPDATE</td>
<td>96</td>
</tr>
<tr>
<td>Considerations for UPDATE</td>
<td>97</td>
</tr>
<tr>
<td>Example of UPDATE</td>
<td>99</td>
</tr>
<tr>
<td>Syntax Description of UPDATE</td>
<td>96</td>
</tr>
<tr>
<td>Considerations for UPDATE</td>
<td>97</td>
</tr>
<tr>
<td>Example of UPDATE</td>
<td>99</td>
</tr>
<tr>
<td>Syntax Description of UPDATE</td>
<td>96</td>
</tr>
<tr>
<td>Considerations for UPDATE</td>
<td>97</td>
</tr>
<tr>
<td>Example of UPDATE</td>
<td>99</td>
</tr>
<tr>
<td>Syntax Description of UPDATE</td>
<td>96</td>
</tr>
<tr>
<td>Considerations for UPDATE</td>
<td>97</td>
</tr>
<tr>
<td>Example of UPDATE</td>
<td>99</td>
</tr>
<tr>
<td>Syntax Description of UPDATE</td>
<td>96</td>
</tr>
<tr>
<td>Considerations for UPDATE</td>
<td>97</td>
</tr>
<tr>
<td>Example of UPDATE</td>
<td>99</td>
</tr>
<tr>
<td>Syntax Description of UPDATE</td>
<td>96</td>
</tr>
<tr>
<td>Considerations for UPDATE</td>
<td>97</td>
</tr>
<tr>
<td>Example of UPDATE</td>
<td>99</td>
</tr>
<tr>
<td>Syntax Description of UPDATE</td>
<td>96</td>
</tr>
<tr>
<td>Considerations for UPDATE</td>
<td>97</td>
</tr>
<tr>
<td>Example of UPDATE</td>
<td>99</td>
</tr>
<tr>
<td>Syntax Description of UPDATE</td>
<td>96</td>
</tr>
<tr>
<td>Considerations for UPDATE</td>
<td>97</td>
</tr>
<tr>
<td>Example of UPDATE</td>
<td>99</td>
</tr>
<tr>
<td>Syntax Description of UPDATE</td>
<td>96</td>
</tr>
<tr>
<td>Considerations for UPDATE</td>
<td>97</td>
</tr>
<tr>
<td>Example of UPDATE</td>
<td>99</td>
</tr>
<tr>
<td>Syntax Description of UPDATE</td>
<td>96</td>
</tr>
<tr>
<td>Considerations for UPDATE</td>
<td>97</td>
</tr>
<tr>
<td>Example of UPDATE</td>
<td>99</td>
</tr>
<tr>
<td>Syntax Description of UPDATE</td>
<td>96</td>
</tr>
<tr>
<td>Considerations for UPDATE</td>
<td>97</td>
</tr>
<tr>
<td>Example of UPDATE</td>
<td>99</td>
</tr>
<tr>
<td>Syntax Description of UPDATE</td>
<td>96</td>
</tr>
<tr>
<td>Considerations for UPDATE</td>
<td>97</td>
</tr>
<tr>
<td>Example of UPDATE</td>
<td>99</td>
</tr>
<tr>
<td>Syntax Description of UPDATE</td>
<td>96</td>
</tr>
<tr>
<td>Considerations for UPDATE</td>
<td>97</td>
</tr>
<tr>
<td>Example of UPDATE</td>
<td>99</td>
</tr>
<tr>
<td>Syntax Description of UPDATE</td>
<td>96</td>
</tr>
<tr>
<td>Considerations for UPDATE</td>
<td>97</td>
</tr>
<tr>
<td>Example of UPDATE</td>
<td>99</td>
</tr>
<tr>
<td>Syntax Description of UPDATE</td>
<td>96</td>
</tr>
<tr>
<td>Considerations for UPDATE</td>
<td>97</td>
</tr>
<tr>
<td>Example of UPDATE</td>
<td>99</td>
</tr>
<tr>
<td>Syntax Description of UPDATE</td>
<td>96</td>
</tr>
<tr>
<td>Considerations for UPDATE</td>
<td>97</td>
</tr>
<tr>
<td>Example of UPDATE</td>
<td>99</td>
</tr>
<tr>
<td>Syntax Description of UPDATE</td>
<td>96</td>
</tr>
<tr>
<td>Considerations for UPDATE</td>
<td>97</td>
</tr>
<tr>
<td>Example of UPDATE</td>
<td>99</td>
</tr>
<tr>
<td>Syntax Description of UPDATE</td>
<td>96</td>
</tr>
<tr>
<td>Considerations for UPDATE</td>
<td>97</td>
</tr>
<tr>
<td>Example of UPDATE</td>
<td>99</td>
</tr>
<tr>
<td>Syntax Description of UPDATE</td>
<td>96</td>
</tr>
<tr>
<td>Considerations for UPDATE</td>
<td>97</td>
</tr>
<tr>
<td>Example of UPDATE</td>
<td>99</td>
</tr>
<tr>
<td>Syntax Description of UPDATE</td>
<td>96</td>
</tr>
<tr>
<td>Considerations for UPDATE</td>
<td>97</td>
</tr>
<tr>
<td>Example of UPDATE</td>
<td>99</td>
</tr>
<tr>
<td>Syntax Description of UPDATE</td>
<td>96</td>
</tr>
<tr>
<td>Considerations for UPDATE</td>
<td>97</td>
</tr>
<tr>
<td>Example of UPDATE</td>
<td>99</td>
</tr>
<tr>
<td>Syntax Description of UPDATE</td>
<td>96</td>
</tr>
<tr>
<td>Considerations for UPDATE</td>
<td>97</td>
</tr>
<tr>
<td>Example of UPDATE</td>
<td>99</td>
</tr>
<tr>
<td>Syntax Description of UPDATE</td>
<td>96</td>
</tr>
<tr>
<td>Considerations for UPDATE</td>
<td>97</td>
</tr>
<tr>
<td>Example of UPDATE</td>
<td>99</td>
</tr>
<tr>
<td>Syntax Description of UPDATE</td>
<td>96</td>
</tr>
<tr>
<td>Considerations for UPDATE</td>
<td>97</td>
</tr>
<tr>
<td>Example of UPDATE</td>
<td>99</td>
</tr>
<tr>
<td>Syntax Description of UPDATE</td>
<td>96</td>
</tr>
<tr>
<td>Considerations for UPDATE</td>
<td>97</td>
</tr>
<tr>
<td>Example of UPDATE</td>
<td>99</td>
</tr>
<tr>
<td>Syntax Description of UPDATE</td>
<td>96</td>
</tr>
<tr>
<td>Considerations for UPDATE</td>
<td>97</td>
</tr>
<tr>
<td>Example of UPDATE</td>
<td>99</td>
</tr>
<tr>
<td>Syntax Description of UPDATE</td>
<td>96</td>
</tr>
<tr>
<td>Considerations for UPDATE</td>
<td>97</td>
</tr>
<tr>
<td>Example of UPDATE</td>
<td>99</td>
</tr>
<tr>
<td>Syntax Description of UPDATE</td>
<td>96</td>
</tr>
<tr>
<td>Considerations for UPDATE</td>
<td>97</td>
</tr>
<tr>
<td>Example of UPDATE</td>
<td>99</td>
</tr>
<tr>
<td>Syntax Description of UPDATE</td>
<td>96</td>
</tr>
<tr>
<td>Considerations for UPDATE</td>
<td>97</td>
</tr>
<tr>
<td>Example of UPDATE</td>
<td>99</td>
</tr>
<tr>
<td>Syntax Description of UPDATE</td>
<td>96</td>
</tr>
<tr>
<td>Considerations for UPDATE</td>
<td>97</td>
</tr>
<tr>
<td>Example of UPDATE</td>
<td>99</td>
</tr>
<tr>
<td>Syntax Description of UPDATE</td>
<td>96</td>
</tr>
<tr>
<td>Considerations for UPDATE</td>
<td>97</td>
</tr>
<tr>
<td>Example of UPDATE</td>
<td>99</td>
</tr>
</tbody>
</table>

3 SQL Language Elements

Authorization IDs.................................................109
Character Sets..................................................110
Columns..........................................................111
Column References.............................................111
Derived Column Names.......................................111
Column Default Settings....................................112
Constraints......................................................113
Creating or Adding Constraints on SQL Tables........113
Correlation Names.............................................114
Explicit Correlation Names.................................114
Implicit Correlation Names................................114
Examples of Correlation Names.........................114
Database Objects..............................................115
### Contents

- Ownership ..........................................................................................................................115
- Database Object Names ......................................................................................................116
- Logical Names for SQL Objects ..........................................................................................116
- SQL Object Namespaces .....................................................................................................116
- Data Types ..........................................................................................................................117
  - Comparable and Compatible Data Types ........................................................................119
  - Character String Data Types ..........................................................................................122
  - Datetime Data Types ........................................................................................................123
  - Interval Data Types ...........................................................................................................125
  - Numeric Data Types .........................................................................................................127
- Expressions ..........................................................................................................................129
  - Character Value Expressions ..........................................................................................129
  - Datetime Value Expressions .............................................................................................130
  - Interval Value Expressions ...............................................................................................133
  - Numeric Value Expressions ..............................................................................................136
- Identifiers ...............................................................................................................................139
  - Regular Identifiers ............................................................................................................139
  - Delimited Identifiers .........................................................................................................139
  - Examples of Identifiers .....................................................................................................139
- Indexes ....................................................................................................................................140
  - SQL Indexes ......................................................................................................................140
- Keys .........................................................................................................................................141
  - Clustering Keys ..................................................................................................................141
  - SYSKEY ............................................................................................................................141
  - Index Keys ........................................................................................................................141
  - Primary Keys ....................................................................................................................141
- Literals ......................................................................................................................................142
  - Character String Literals ...................................................................................................142
  - Datetime Literals ...............................................................................................................144
  - Interval Literals ..................................................................................................................145
  - Numeric Literals ...............................................................................................................147
- Null ..........................................................................................................................................149
  - Using Null Versus Default Values ....................................................................................149
  - Defining Columns That Allow or Prohibit Null ..................................................................149
- Predicates ................................................................................................................................151
  - BETWEEN Predicate ..........................................................................................................151
  - Comparison Predicates .....................................................................................................152
  - EXISTS Predicate ............................................................................................................156
  - IN Predicate .....................................................................................................................157
  - LIKE Predicate ..................................................................................................................159
  - NULL Predicate ................................................................................................................161
  - Quantified Comparison Predicates ....................................................................................162
- Schemas ....................................................................................................................................165
- Search Condition .....................................................................................................................166
  - Considerations for Search Condition ................................................................................166
  - Examples of Search Condition .........................................................................................166
- Subquery ..................................................................................................................................168
  - SELECT Form of a Subquery ...............................................................................................168
  - Using Subqueries to Provide Comparison Values .............................................................168
  - Nested Subqueries When Providing Comparison Values .................................................168
  - Correlated Subqueries When Providing Comparison Values .........................................168
- Tables .......................................................................................................................................170
  - Base Tables and Views .......................................................................................................170
  - Example of a Base Table .....................................................................................................170
- Views .......................................................................................................................................171
Valid Conversions for CAST ................................................................. 214
Examples of CAST.............................................................................. 215
CEILING Function............................................................................. 216
Example of CEILING......................................................................... 216
CHAR Function.................................................................................. 217
Considerations for CHAR................................................................. 217
Example of CHAR............................................................................... 217
CHAR_LENGTH Function................................................................. 218
Considerations for CHAR_LENGTH.................................................... 218
Examples of CHAR_LENGTH............................................................ 218
COALESCE Function........................................................................ 219
Example of COALESCE..................................................................... 219
CODE_VALUE Function..................................................................... 220
Example of CODE_VALUE Function.................................................. 220
CONCAT Function............................................................................ 221
Concatenation Operator (||)............................................................... 221
Considerations for CONCAT............................................................. 221
Examples of CONCAT........................................................................ 221
CONVERTTOHEX Function.............................................................. 223
Considerations for CONVERTTOHEX............................................. 223
Examples of CONVERTTOHEX....................................................... 223
CONVERTTIMESTAMP Function....................................................... 225
Considerations for CONVERTTIMESTAMP..................................... 225
Examples of CONVERTTIMESTAMP.............................................. 225
COS Function................................................................................... 226
Example of COS................................................................................ 226
COSH Function................................................................................ 227
Example of COSH........................................................................... 227
COUNT Function............................................................................... 228
Considerations for COUNT............................................................. 228
Examples of COUNT......................................................................... 228
CURRENT Function......................................................................... 230
Example of CURRENT.................................................................... 230
CURRENT_DATE Function............................................................... 231
Examples of CURRENT_DATE......................................................... 231
CURRENT_TIME Function.............................................................. 232
Example of CURRENT_TIME......................................................... 232
CURRENT_TIMESTAMP Function.................................................... 233
Example of CURRENT_TIMESTAMP.............................................. 233
DATE_ADD Function......................................................................... 234
Examples of DATE_ADD................................................................. 234
DATE_SUB Function......................................................................... 235
Examples of DATE_SUB................................................................. 235
DATEADD Function.......................................................................... 236
Examples of DATEADD................................................................. 236
DATEDIFF Function.......................................................................... 237
Examples of DATEDIFF................................................................. 237
DATEFORMAT Function................................................................. 238
Considerations for DATEFORMAT................................................. 238
Examples of DATEFORMAT.......................................................... 238
DATE_PART Function [of an Interval].............................................. 239
Examples of DATE_PART............................................................... 239
DATE_PART Function [of a Timestamp].......................................... 240
Examples of DATE_PART............................................................... 240
DATE_TRUNC Function..................................................................... 241
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>DAY Function</td>
<td>242</td>
</tr>
<tr>
<td>Example of DAY</td>
<td>242</td>
</tr>
<tr>
<td>DAYNAME Function</td>
<td>243</td>
</tr>
<tr>
<td>Considerations for DAYNAME</td>
<td>243</td>
</tr>
<tr>
<td>Example of DAYNAME</td>
<td>243</td>
</tr>
<tr>
<td>DAYOFMONTH Function</td>
<td>244</td>
</tr>
<tr>
<td>Examples of DAYOFMONTH</td>
<td>244</td>
</tr>
<tr>
<td>DAYOFWEEK Function</td>
<td>245</td>
</tr>
<tr>
<td>Example of DAYOFWEEK</td>
<td>245</td>
</tr>
<tr>
<td>DAYOFYEAR Function</td>
<td>246</td>
</tr>
<tr>
<td>Example of DAYOFYEAR</td>
<td>246</td>
</tr>
<tr>
<td>DECODE Function</td>
<td>247</td>
</tr>
<tr>
<td>Considerations for DECODE</td>
<td>247</td>
</tr>
<tr>
<td>Examples of DECODE</td>
<td>248</td>
</tr>
<tr>
<td>DEGREES Function</td>
<td>250</td>
</tr>
<tr>
<td>Examples of DEGREES</td>
<td>250</td>
</tr>
<tr>
<td>DIFF1 Function</td>
<td>251</td>
</tr>
<tr>
<td>Considerations for DIFF1</td>
<td>251</td>
</tr>
<tr>
<td>Examples of DIFF1</td>
<td>251</td>
</tr>
<tr>
<td>DIFF2 Function</td>
<td>253</td>
</tr>
<tr>
<td>Considerations for DIFF2</td>
<td>253</td>
</tr>
<tr>
<td>Examples of DIFF2</td>
<td>253</td>
</tr>
<tr>
<td>EXP Function</td>
<td>255</td>
</tr>
<tr>
<td>Examples of EXP</td>
<td>255</td>
</tr>
<tr>
<td>EXPLAIN Function</td>
<td>256</td>
</tr>
<tr>
<td>Considerations for EXPLAIN</td>
<td>256</td>
</tr>
<tr>
<td>Function</td>
<td>258</td>
</tr>
<tr>
<td>Examples of EXPLAIN Function</td>
<td>258</td>
</tr>
<tr>
<td>EXTRACT Function</td>
<td>259</td>
</tr>
<tr>
<td>Examples of EXTRACT</td>
<td>259</td>
</tr>
<tr>
<td>FLOOR Function</td>
<td>260</td>
</tr>
<tr>
<td>Examples of FLOOR</td>
<td>260</td>
</tr>
<tr>
<td>HOUR Function</td>
<td>261</td>
</tr>
<tr>
<td>Example of HOUR</td>
<td>261</td>
</tr>
<tr>
<td>INSERT Function</td>
<td>262</td>
</tr>
<tr>
<td>Examples of INSERT</td>
<td>262</td>
</tr>
<tr>
<td>ISNULL Function</td>
<td>263</td>
</tr>
<tr>
<td>Examples of ISNULL</td>
<td>263</td>
</tr>
<tr>
<td>JULIANTIMESTAMP Function</td>
<td>264</td>
</tr>
<tr>
<td>Considerations for JULIANTIMESTAMP</td>
<td>264</td>
</tr>
<tr>
<td>Examples of JULIANTIMESTAMP</td>
<td>264</td>
</tr>
<tr>
<td>LASTNOTNULL Function</td>
<td>265</td>
</tr>
<tr>
<td>Example of LASTNOTNULL</td>
<td>265</td>
</tr>
<tr>
<td>LCASE Function</td>
<td>266</td>
</tr>
<tr>
<td>Example of LCASE</td>
<td>266</td>
</tr>
<tr>
<td>LEFT Function</td>
<td>267</td>
</tr>
<tr>
<td>Examples of LEFT</td>
<td>267</td>
</tr>
<tr>
<td>LOCATE Function</td>
<td>268</td>
</tr>
<tr>
<td>Considerations for LOCATE</td>
<td>268</td>
</tr>
<tr>
<td>Examples of LOCATE</td>
<td>268</td>
</tr>
<tr>
<td>LOG Function</td>
<td>269</td>
</tr>
<tr>
<td>Example of LOG</td>
<td>269</td>
</tr>
<tr>
<td>LOG10 Function</td>
<td>270</td>
</tr>
<tr>
<td>Example of LOG10</td>
<td>270</td>
</tr>
<tr>
<td>LOWER Function</td>
<td>271</td>
</tr>
</tbody>
</table>

Contents 9
MDAM_SCAN_METHOD...........................................................................................................351
SUBQUERY_UNNESTING........................................................................................................351
Managing Schemas..........................................................................................................351
SCHEMA..........................................................................................................................351
Transaction Control and Locking....................................................................................352
BLOCK_TO_PREVENT_HALLOWEEN..............................................................................352
UPD_ORDERED.................................................................................................................353

C Limits...........................................................................................................................354
Index...............................................................................................................................355
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 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. 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 corrections to “Using Trafodion SQL to Access HBase Tables” (page 18).

Document Organization

<table>
<thead>
<tr>
<th>Chapter or Appendix</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapter 1: “Introduction” (page 18)</td>
<td>Introduces Trafodion SQL and covers topics such as data consistency, transaction management, and ANSI compliance.</td>
</tr>
<tr>
<td>Chapter 2: “SQL Statements” (page 24)</td>
<td>Describes the SQL statements supported by Trafodion SQL.</td>
</tr>
<tr>
<td>Chapter 3: “SQL Language Elements” (page 108)</td>
<td>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.</td>
</tr>
<tr>
<td>Chapter 4: “SQL Clauses” (page 172)</td>
<td>Describes clauses used by Trafodion SQL statements.</td>
</tr>
<tr>
<td>Chapter 5: “SQL Functions and Expressions” (page 194)</td>
<td>Describes specific functions and expressions that you can use in Trafodion SQL statements.</td>
</tr>
<tr>
<td>Appendix A: “Reserved Words” (page 344)</td>
<td>Lists the words that are reserved in Trafodion SQL.</td>
</tr>
<tr>
<td>Appendix B: “Control Query Default (CQD) Attributes” (page 348)</td>
<td>Describes the Control Query Default (CQD) attributes that are supported in a Trafodion SQL environment.</td>
</tr>
<tr>
<td>Appendix C: “Limits” (page 354)</td>
<td>Describes limits in Trafodion SQL.</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Part Number</th>
<th>Product Version</th>
<th>Publication Date</th>
</tr>
</thead>
</table>
| T775-080-004 | Trafodion Release 0.8.0 Beta  
This edition includes corrections to “Using Trafodion SQL to Access HBase Tables” (page 18). | December 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 36). | August 2014 |
<table>
<thead>
<tr>
<th>Part Number</th>
<th>Product Version</th>
<th>Publication Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>T775-080-002</td>
<td>Trafodion Release 0.8.0 Beta</td>
<td>August 2014</td>
</tr>
<tr>
<td></td>
<td>This edition of the manual includes updates to address Launchpad bug 1352479. See the “SELECT Statement” (page 74).</td>
<td></td>
</tr>
<tr>
<td>T775-080-001</td>
<td>Trafodion Release 0.8.0 Beta</td>
<td>June 2014</td>
</tr>
</tbody>
</table>

**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. This introduction describes:

- “SQL Language”
- “Using Trafodion SQL to Access HBase 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 24).

SQL language elements are part of statements and commands and include data types, expressions, functions, identifiers, literals, and predicates. For more information, see Chapter 3: “SQL Language Elements” (page 108) and Chapter 4: “SQL Clauses” (page 172). For information on specific functions and expressions, see Chapter 5: “SQL Functions and Expressions” (page 194).

Using Trafodion SQL to Access HBase Tables

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

- “Initializing the Trafodion Metadata” (page 18)
- “Accessing Trafodion SQL Tables” (page 18)
- “Trafodion SQL Tables Versus Native HBase Tables” (page 19)
- “Supported SQL Statements With HBase Tables” (page 19)

For a list of Control Query Default (CQD) settings for the HBase environment, see “HBase Environment CQDs” (page 348).

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;

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 19) and “Tables” (page 170).

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 36).

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 74)
- “INSERT Statement” (page 63)
- “UPDATE Statement” (page 96)
- “DELETE Statement” (page 51)
- “MERGE Statement” (page 68)
- “GET Statement” (page 61)
- “INVOKE Statement” (page 67)
- “CREATE TABLE Statement” (page 36)
- “CREATE INDEX Statement” (page 34)
- “CREATE VIEW Statement” (page 47)
- “DROP INDEX Statement” (page 53)
- “DROP TABLE Statement” (page 54)
- “DROP VIEW Statement” (page 55)
- “ALTER TABLE Statement” (page 26)

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 21).

**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 20)
- “Rules for DML Statements” (page 21)
- “Effect of AUTOCOMMIT Option” (page 21)
- “Concurrency” (page 21)
- “Transaction Isolation Levels” (page 21)

**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 93). 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 93).

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 19).

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
- COMMIT WORK statement
- CREATE TABLE statement
- CREATE VIEW statement
- DELETE statement
- DROP TABLE statement
- DROP VIEW statement
- EXECUTE statement
- INSERT statement
- MERGE statement
- PREPARE 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.

- BEGIN WORK statement
- CONTROL QUERY DEFAULT statement
- CREATE INDEX statement
- DROP INDEX statement
- EXPLAIN statement
- UPDATE STATISTICS 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”
- “Control Statements”
- “Object Naming Statements”
- “Transaction Control Statements”

Data Definition Language (DDL) Statements

Use these DDL statements to define, delete, or modify the definition of a Trafodion SQL schema, or object, or the authorization to use an 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.

<table>
<thead>
<tr>
<th>Statement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALTER TABLE Statement (page 26)</td>
<td>Changes a table.</td>
</tr>
<tr>
<td>CREATE INDEX Statement (page 34)</td>
<td>Creates an index on a table.</td>
</tr>
<tr>
<td>CREATE TABLE Statement (page 36)</td>
<td>Creates a table.</td>
</tr>
<tr>
<td>CREATE VIEW Statement (page 47)</td>
<td>Creates a view.</td>
</tr>
<tr>
<td>DROP INDEX Statement (page 53)</td>
<td>Drops an index.</td>
</tr>
<tr>
<td>DROP TABLE Statement (page 54)</td>
<td>Drops a table.</td>
</tr>
<tr>
<td>DROP VIEW Statement (page 55)</td>
<td>Drops a view.</td>
</tr>
<tr>
<td>INVOKE Statement (page 67)</td>
<td>Generates a record description that corresponds to a row in the specified table or view.</td>
</tr>
</tbody>
</table>

Data Manipulation Language (DML) Statements

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

<table>
<thead>
<tr>
<th>Statement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DELETE Statement (page 51)</td>
<td>Deletes rows from a table or view.</td>
</tr>
<tr>
<td>INSERT Statement (page 63)</td>
<td>Inserts data into tables and views.</td>
</tr>
<tr>
<td>MERGE Statement (page 68)</td>
<td>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.</td>
</tr>
<tr>
<td>SELECT Statement (page 74)</td>
<td>Retrieves data from tables and views.</td>
</tr>
<tr>
<td>TABLE Statement (page 95)</td>
<td>Equivalent to the query specification SELECT * FROM table</td>
</tr>
</tbody>
</table>


### Control Statements

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

- **“CONTROL QUERY DEFAULT Statement” (page 33)**: Changes a default attribute to influence a query plan.

### Object Naming Statements

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

- **“SET SCHEMA Statement” (page 92)**: Sets the default ANSI schema for unqualified object names for the current session.

### Transaction Control Statements

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

- **“BEGIN WORK Statement” (page 31)**: Starts a transaction.
- **“COMMIT WORK Statement” (page 32)**: Commits changes made during a transaction and ends the transaction.
- **“ROLLBACK WORK Statement” (page 73)**: Undoes changes made during a transaction and ends the transaction.
- **“SET TRANSACTION Statement” (page 93)**: Sets attributes for the next SQL transaction — whether to automatically commit database changes.
The ALTER TABLE statement changes a Trafodion SQL table. See “Database Object Names” (page 116).

**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]
  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[GER] [SIGNED|UNSIGNED]
  | LARGEINT
  | DECIMAL [(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 }
```
### Syntax Description of ALTER TABLE

**name**
- specifies the current name of the object. See “Database Object Names” (page 116).

**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 139).

  **data-type**
  - specifies the data type of the values that can be stored in column-name. See “Data Types” (page 117). 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.

  If 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 116).

  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 149).

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 166). 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:

referred-table [(column-list)]

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

column-list specifies the column or set of columns in the referred-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 116).

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 166). 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 116).

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...

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

To alter a table, you must own the table or be the schema owner.
ALTER TABLE works only on user-created tables.

Example of ALTER TABLE

This example adds a column:

```sql
ALTER TABLE persnl.project
ADD COLUMN projlead
  NUMERIC (4) UNSIGNED
```
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 20). BEGIN WORK will raise an error if a transaction is currently active. BEGIN WORK is a Trafodion SQL extension.

**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:

```sql
--- 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.
```
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 20).

```
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 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 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 348).
- `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:

```sql
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:
  ```sql
  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:
  ```sql
  CONTROL QUERY DEFAULT HBASE_MAX_COLUMN_NAME_LENGTH RESET;
  ```
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 “Database Object Names” (page 116).

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 [ASCENDING | DESCENDING]
    [, column-name [ASCENDING | DESCENDING]]...)
```

**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 names. 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 116).

*column-name [ASCENDING | DESCENDING] [, column-name [ASCENDING | DESCENDING]]...*

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.

**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. To create an SQL index, you must have one of the following privileges:

- Be the table owner.
- Be the schema owner.

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.

**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 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 a 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 “Database Object Names” (page 116).

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)}]
   [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:
   CHARACTER [(length [CHARACTERS])]
      [CHARACTER SET char-set-name]
      [UPSHIFT] [[NOT]CASESPECIFIC]
   | CHAR 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
   | DECIMAL [(precision [,scale])] [SIGNED|UNSIGNED]
   | 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]

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 40).

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
  is the ANSI logical name for the new table and 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.

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 44).

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 44).

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 117). 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 44).

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 173).

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 116).

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 149).

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 166).

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 PARTITION file-option, you cannot specify WITH PARTITIONS.

Considerations for CREATE TABLE
The following subsections provide considerations for various CREATE TABLE options:

• “Considerations for CREATE VOLATILE TABLE” (page 40)
• “Considerations for CREATE TABLE ... LIKE” (page 43)
• “Considerations for LOAD IF EXISTS and NO LOAD options of CREATE TABLE AS” (page 44)
• “Considerations for CREATE TABLE AS” (page 43)

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 44).
• 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 41).
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

<table>
<thead>
<tr>
<th>Precedence of Data Types (From Low to High)</th>
</tr>
</thead>
<tbody>
<tr>
<td>VARCHAR</td>
</tr>
<tr>
<td>INTERVAL</td>
</tr>
<tr>
<td>DATETIME</td>
</tr>
<tr>
<td>CHAR(ACTER)</td>
</tr>
<tr>
<td>DECIMAL (signed, unsigned)</td>
</tr>
<tr>
<td>SMALLINT (signed, unsigned)</td>
</tr>
<tr>
<td>INTEGER (signed, unsigned)</td>
</tr>
<tr>
<td>LARGEINT (signed only)</td>
</tr>
</tbody>
</table>

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:

```sql
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:

```sql
>>>create volatile table t (a int, primary key(a));
--- SQL operation complete.

Only one unique null value is allowed:

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

```sql
>>>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 41):

- Selects column `a` as the primary and partitioning key:
  ```sql
  create volatile table t (a int);
  ```

- Selects column `b` because `int` has a higher precedence than `char`:
  ```sql
  create volatile table t (a char(10), b int);
  ```

- Selects column `b` because `not null` has precedence over nullable columns:
  ```sql
  create volatile table t (a int, b int not null);
  ```

- Selects column `b` because `int` has precedence over `decimal`:
  ```sql
  create volatile table t (a decimal(10), b int);
  ```

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

• `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 171).

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

Syntax Description of CREATE VIEW

`view` specifies the name of the view. See “Database Object Names” (page 116).

`(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 142). For the syntax and syntax description of `query-expr`, see “SELECT Statement” (page 74). 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 74). 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

- 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 create a view, you must be the owner of the schemas in the view or be the owner of the objects in the view.

When you create a view on a single table, the owner of the view is automatically given all privileges on the view. However, when you create a view that spans multiple tables, the owner of the view is given only SELECT privileges.

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 creates a view with a column list:
  ```
  CREATE VIEW SALES.MYVIEW2
  (v_ordernum, t_partnum) AS
  ```
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.

<table>
<thead>
<tr>
<th>Searched DELETE is:</th>
</tr>
</thead>
<tbody>
<tr>
<td>DELETE FROM table</td>
</tr>
<tr>
<td>[WHERE search-condition ]</td>
</tr>
<tr>
<td>[[FOR] access-option ACCESS]</td>
</tr>
</tbody>
</table>

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 116).

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 166).

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 19).

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:
  
  ```sql
  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:
  
  ```sql
  DELETE FROM sales.orders
  WHERE salesrep = 220 AND custnum <> 1234;
  ```
  
  --- 2 row(s) deleted.

- Remove all suppliers not in Texas from the table PARTSUPP:
  
  ```sql
  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:
  
  ```sql
  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:
  
  ```sql
  delete from table1 where a in
  (select a from table1 where b > 200)
  ```
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 “Database Object Names” (page 116). 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.

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

**Syntax Description of DROP INDEX**

`index` is the index to drop.

For information, see “Database Object Names” (page 116).

**Considerations for DROP INDEX**

**Authorization and Availability Requirements**

To drop an index, you must be the owner of the table associated with the index.

**Examples of DROP INDEX**

- This example drops an index:
  
```sql
  DROP INDEX myindex;
  ```

- This example drops a volatile index:
  
```sql
  DROP VOLATILE INDEX vindex;
  ```
**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 “Database Object Names” (page 116).

**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 drop a table, you must be the table owner or the schema owner.

**Examples of DROP TABLE**

- This example drops a table:
  ```sql
  DROP TABLE mysch.mytable;
  ```

- This example drops a volatile table:
  ```sql
  DROP VOLATILE TABLE vtable;
  ```
The DROP VIEW statement deletes a Trafodion SQL view. See “Views” (page 171).

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 and Availability Requirements
To drop a view, you must be the owner of the view or owner of the schema.

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 139).

*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 71). 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 256).

EXPLAIN [OPTIONS {'f'}] {FOR QID query-text | prepared-stmt-name}

Table 2 EXPLAIN Statement Options

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Option Type</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPTIONS 'f'</td>
<td>Formatted</td>
<td>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.</td>
</tr>
</tbody>
</table>

Plans displayed by the EXPLAIN statement are ordered from top (root operator) to bottom (leaf operators).

Syntax Description of EXPLAIN

formatted. See “Formatted [OPTIONS ‘f’] Considerations” (page 60).

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

- “Case Considerations” (page 59)
- “Number Considerations” (page 60)
- “Formatted [OPTIONS ‘f’] Considerations” (page 60)

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. To enable this feature, contact your HP Support representative for assistance.

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:

<table>
<thead>
<tr>
<th>LC</th>
<th>RC</th>
<th>OP</th>
<th>OPERATOR</th>
<th>OPT</th>
<th>DESCRIPTION</th>
<th>CARD</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>.</td>
<td>2</td>
<td>root</td>
<td></td>
<td></td>
<td>1.00E+002</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td>1</td>
<td>trafodion_scan</td>
<td>REGION</td>
<td></td>
<td>1.00E+002</td>
</tr>
</tbody>
</table>

--- 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 the database objects that exist in the Trafodion instance.

<table>
<thead>
<tr>
<th>GET option</th>
</tr>
</thead>
<tbody>
<tr>
<td>option is:</td>
</tr>
<tr>
<td>SCHEMAS [IN CATALOG catalog-name]</td>
</tr>
<tr>
<td>TABLES [IN SCHEMA [catalog-name.]schema-name]</td>
</tr>
<tr>
<td>VIEWS [IN SCHEMA [catalog-name.]schema-name]</td>
</tr>
<tr>
<td>VIEWS ON TABLE [[catalog-name.]schema-name.]table-name</td>
</tr>
</tbody>
</table>

Syntax Description of GET

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.

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.

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

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".

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 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;
The INSERT statement is a DML statement that inserts rows into a table or view.

**Syntax Description of INSERT**

```sql
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
```

**Table**

- `INSERT INTO` specifies the table and column(s) to insert data into.
- `table` is the name of the table or view.
- `(target-col-list)` specifies the columns to insert data into.
- `insert-source` specifies the rows of data to insert.

**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 66)). 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 93).

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`:

  ```sql
  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:

  ```sql
  SELECT * FROM sales.customer
  WHERE custnum = 4777;
  
  CUSTNUM  CUSTNAME       STREET         ... POSTCODE  CREDIT
  -------  --------------  --------------     --------  ------
  4777    ZYROTECHNIKS   11211 40TH ST. ... 01803     A2
  
  --- 1 row(s) selected.
  
  - Insert a row into the CUSTOMER table using a `target-col-list`:

  ```sql
  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:

  ```sql
  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.
INVOKED Statement

- “Syntax Description of 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.

INVOKED is a Trafodion SQL extension.

**Syntax Description of INVOKE**

```
INVOKED table-name
```

table-name specifies the name of a table, view, or index for which to generate a record description. See “Tables” (page 170).

**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**

**Upset 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 $t$.

```sql
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.

```sql
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.

```sql
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.

```sql
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:

```sql
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 68).

**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);
```
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 139). 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
  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>
  
  SQL> 
  ```

- Prepare a SELECT statement with an unnamed parameter (?) and later run EXECUTE on it:

  ```sql
  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 56).
ROLLBACK WORK Statement

- “Syntax Description of ROLLBACK WORK” (page 73)
- “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 20).

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.
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
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]

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])
  | (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)]
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 \textit{corr} of the table references, as specified in the FROM clause. See “Correlation Names” (page 114).

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.

corr\.single-col \[ [AS] name \]

specifies a column.

corr\.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 with a \textit{name}.

See the discussion of limitations in “Considerations for Select List” (page 84).

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 114).

col-expr [(AS] name [,col-expr [(AS] name)]...

specifies the items in col-expr-list, a list of derived columns.

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 51).

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.

[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.

(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 96).

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:
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.

[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.

For example:
RETURN old.empno, old.salary, new.salary,
(\(\text{new.salary - old.salary}\))
AS emp (empno, oldsalary, newsalary, increase).

(insert-statement) [AS] corr [[col-expr-list]]
For the syntax of insert-statement, see the “INSERT Statement” (page 63).

[AS] corr [[col-expr-list]]
specifies a correlation name, corr, and an optional column list.

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 166). 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 168).

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 177).

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 187).

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 84).

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 166).

[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 19).

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 84).

ORDER BY {colname | colnum} [ASCENDING | DESCENDING] [, {colname | colnum} [ASCENDING | DESCENDING]]...

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 84).

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.
NOTE: HP recommends 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.

Examples of Multiple Distinct Aggregates
- This statement contains distinct aggregates:

  ```sql
  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.

  ```sql
  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:

  ```sql
  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 194).
- 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;

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;
  ```

<table>
<thead>
<tr>
<th>JOBCODE</th>
<th>DEPTNUM</th>
<th>FIRST_NAME</th>
<th>LAST_NAME</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>600</td>
<td>1500</td>
<td>JONATHAN</td>
<td>MITCHELL</td>
<td>32000.00</td>
</tr>
<tr>
<td>600</td>
<td>1500</td>
<td>JIMMY</td>
<td>SCHNEIDER</td>
<td>26000.00</td>
</tr>
<tr>
<td>900</td>
<td>2500</td>
<td>MIRIAM</td>
<td>KING</td>
<td>18000.00</td>
</tr>
<tr>
<td>900</td>
<td>1000</td>
<td>SUE</td>
<td>CRAMER</td>
<td>19000.00</td>
</tr>
</tbody>
</table>
  ...     

- 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;
  ```

<table>
<thead>
<tr>
<th>JOBCODE</th>
<th>EXPR</th>
</tr>
</thead>
<tbody>
<tr>
<td>600</td>
<td>29000.00</td>
</tr>
<tr>
<td>900</td>
<td>25100.00</td>
</tr>
</tbody>
</table>
  ... 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);
  ```

<table>
<thead>
<tr>
<th>JOBDESC</th>
<th>FIRST_NAME</th>
<th>LAST_NAME</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>SALESREP</td>
<td>TIM</td>
<td>WALKER</td>
<td>32000.00</td>
</tr>
<tr>
<td>SALESREP</td>
<td>HERBERT</td>
<td>KARAJAN</td>
<td>29000.00</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>ENGINEER</td>
<td>MARK</td>
<td>FOLEY</td>
<td>33000.00</td>
</tr>
<tr>
<td>ENGINEER</td>
<td>MARIA</td>
<td>JOSEF</td>
<td>18000.10</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>SECRETARY</td>
<td>BILL</td>
<td>WINN</td>
<td>32000.00</td>
</tr>
<tr>
<td>SECRETARY</td>
<td>DINAH</td>
<td>CLARK</td>
<td>37000.00</td>
</tr>
</tbody>
</table>
  ...
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.

<table>
<thead>
<tr>
<th>EMPLOYEE Table</th>
<th>JOB Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMPNUM ...</td>
<td>JOBCODE ...</td>
</tr>
</tbody>
</table>

2. Drop rows with unequal job codes.

<table>
<thead>
<tr>
<th>EMPLOYEE Table</th>
<th>JOB Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMPNUM ...</td>
<td>JOBCODE ...</td>
</tr>
<tr>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>75</td>
<td>300</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>178</td>
<td>900</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>207</td>
<td>420</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>568</td>
<td>300</td>
</tr>
</tbody>
</table>

3. Drop rows with job codes not equal to 900, 300, or 420.

<table>
<thead>
<tr>
<th>EMPLOYEE Table</th>
<th>JOB Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMPNUM ...</td>
<td>JOBCODE ...</td>
</tr>
<tr>
<td>75</td>
<td>300</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>178</td>
<td>900</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>207</td>
<td>420</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>568</td>
<td>300</td>
</tr>
</tbody>
</table>
4. Process the select list, leaving only four columns.

<table>
<thead>
<tr>
<th>JOBDESC</th>
<th>FIRST_NAME</th>
<th>LAST_NAME</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>SALESREP</td>
<td>TIM</td>
<td>WALKER</td>
<td>32000</td>
</tr>
<tr>
<td>. . .</td>
<td>. . .</td>
<td>. . .</td>
<td>. . .</td>
</tr>
<tr>
<td>SECRETARY</td>
<td>JOHN</td>
<td>CHOU</td>
<td>28000</td>
</tr>
<tr>
<td>. . .</td>
<td>. . .</td>
<td>. . .</td>
<td>. . .</td>
</tr>
<tr>
<td>ENGINEER</td>
<td>MARK</td>
<td>FOLEY</td>
<td>33000</td>
</tr>
<tr>
<td>. . .</td>
<td>. . .</td>
<td>. . .</td>
<td>. . .</td>
</tr>
<tr>
<td>SALESREP</td>
<td>JESSICA</td>
<td>CRINER</td>
<td>39500</td>
</tr>
</tbody>
</table>

The final result is shown in the output:

<table>
<thead>
<tr>
<th>JOBDESC</th>
<th>FIRST_NAME</th>
<th>LAST_NAME</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>SALESREP</td>
<td>TIM</td>
<td>WALKER</td>
<td>32000.00</td>
</tr>
<tr>
<td>. . .</td>
<td>. . .</td>
<td>. . .</td>
<td>. . .</td>
</tr>
<tr>
<td>SECRETARY</td>
<td>JOHN</td>
<td>CHOU</td>
<td>28000.00</td>
</tr>
<tr>
<td>. . .</td>
<td>. . .</td>
<td>. . .</td>
<td>. . .</td>
</tr>
</tbody>
</table>

- 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;
```

<table>
<thead>
<tr>
<th>JOBCODE</th>
<th>DEPTNUM</th>
<th>(EXPR)</th>
<th>(EXPR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>900</td>
<td>1500</td>
<td>17000.00</td>
<td>17000.00</td>
</tr>
<tr>
<td>900</td>
<td>2500</td>
<td>18000.00</td>
<td>18000.00</td>
</tr>
<tr>
<td>. . .</td>
<td>. . .</td>
<td>. . .</td>
<td>. . .</td>
</tr>
<tr>
<td>300</td>
<td>3000</td>
<td>19000.00</td>
<td>32000.00</td>
</tr>
<tr>
<td>900</td>
<td>2000</td>
<td>32000.00</td>
<td>32000.00</td>
</tr>
<tr>
<td>. . .</td>
<td>. . .</td>
<td>. . .</td>
<td>. . .</td>
</tr>
<tr>
<td>300</td>
<td>3200</td>
<td>22000.00</td>
<td>33000.10</td>
</tr>
<tr>
<td>420</td>
<td>4000</td>
<td>18000.10</td>
<td>36000.00</td>
</tr>
<tr>
<td>. . .</td>
<td>. . .</td>
<td>. . .</td>
<td>. . .</td>
</tr>
</tbody>
</table>

--- 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;
```

<table>
<thead>
<tr>
<th>Order/Num</th>
<th>Part/Num</th>
<th>Unit/Price</th>
<th>Qty/Ord</th>
<th>Part/Num</th>
<th>Part Description</th>
<th>PRICE</th>
<th>Qty/Avail</th>
</tr>
</thead>
<tbody>
<tr>
<td>400410</td>
<td>212</td>
<td>2450.00</td>
<td>12</td>
<td>212</td>
<td>PCSILVER, 20 MB</td>
<td>2500.00</td>
<td>3525</td>
</tr>
</tbody>
</table>
• 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:

```sql
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:

```sql
SELECT * FROM job UNION SELECT * FROM job_corporate;
```

<table>
<thead>
<tr>
<th>JOBCODE</th>
<th>JOBDESC</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>MANAGER</td>
</tr>
<tr>
<td>200</td>
<td>PRODUCTION SUPV</td>
</tr>
<tr>
<td>250</td>
<td>ASSEMBLER</td>
</tr>
<tr>
<td>300</td>
<td>SALESREP</td>
</tr>
<tr>
<td>400</td>
<td>SYSTEM ANALYST</td>
</tr>
<tr>
<td>420</td>
<td>ENGINEER</td>
</tr>
<tr>
<td>450</td>
<td>PROGRAMMER</td>
</tr>
<tr>
<td>500</td>
<td>ACCOUNTANT</td>
</tr>
<tr>
<td>600</td>
<td>ADMINISTRATOR</td>
</tr>
<tr>
<td>900</td>
<td>SECRETARY</td>
</tr>
<tr>
<td>100</td>
<td>CORP MANAGER</td>
</tr>
<tr>
<td>300</td>
<td>CORP SALESREP</td>
</tr>
<tr>
<td>400</td>
<td>CORP SYSTEM ANALYS</td>
</tr>
<tr>
<td>500</td>
<td>CORP ACCOUNTANT</td>
</tr>
<tr>
<td>600</td>
<td>CORP ADMINISTRATOR</td>
</tr>
<tr>
<td>900</td>
<td>CORP SECRETARY</td>
</tr>
</tbody>
</table>

--- 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:

```sql
SELECT *
FROM employee
FULL OUTER JOIN
department
ON employee.DepartmentID = department.DepartmentID;
```

<table>
<thead>
<tr>
<th>LastName</th>
<th>DepartmentID</th>
<th>DepartmentName</th>
<th>DepartmentID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smith</td>
<td>34</td>
<td>Clerical</td>
<td>34</td>
</tr>
<tr>
<td>Jones</td>
<td>33</td>
<td>Engineering</td>
<td>33</td>
</tr>
<tr>
<td>Robinson</td>
<td>34</td>
<td>Clerical</td>
<td>34</td>
</tr>
<tr>
<td>Jasper</td>
<td>36</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>Steinberg</td>
<td>33</td>
<td>Engineering</td>
<td>33</td>
</tr>
</tbody>
</table>
Present two ways to select the same data submitted by customers from California.

The first way:

```sql
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;
```

<table>
<thead>
<tr>
<th>ORDERNUM</th>
<th>(EXPR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>200490</td>
<td>1030.00</td>
</tr>
<tr>
<td>300350</td>
<td>71025.00</td>
</tr>
<tr>
<td>300380</td>
<td>28560.00</td>
</tr>
</tbody>
</table>

--- 3 row(s) selected.

The second way:

```sql
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;
```

<table>
<thead>
<tr>
<th>ORDERNUM</th>
<th>(EXPR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>200490</td>
<td>1030.00</td>
</tr>
<tr>
<td>300350</td>
<td>71025.00</td>
</tr>
<tr>
<td>300380</td>
<td>28560.00</td>
</tr>
</tbody>
</table>

--- 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:

```sql
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;
```

<table>
<thead>
<tr>
<th>Employee/Number</th>
<th>First Name</th>
<th>Last Name</th>
<th>salary</th>
<th>(EXPR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>209</td>
<td>SUSAN</td>
<td>CHAPMAN</td>
<td>17000.00</td>
<td>.61</td>
</tr>
<tr>
<td>235</td>
<td>MIRIAM</td>
<td>KING</td>
<td>18000.00</td>
<td>.65</td>
</tr>
<tr>
<td>224</td>
<td>MARIA</td>
<td>JOSEF</td>
<td>18000.10</td>
<td>.65</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>23</td>
<td>JERRY</td>
<td>HOWARD</td>
<td>137000.10</td>
<td>4.94</td>
</tr>
<tr>
<td>32</td>
<td>THOMAS</td>
<td>RUDLOFF</td>
<td>138000.40</td>
<td>4.98</td>
</tr>
<tr>
<td>1</td>
<td>ROGER</td>
<td>GREEN</td>
<td>175500.00</td>
<td>6.33</td>
</tr>
</tbody>
</table>

--- 62 row(s) selected.

Examples of using expressions in the GROUP BY clause:

```sql
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”
- “Consideration 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.

```sql
SET SCHEMA default-schema-name
```

Syntax Description of SET SCHEMA

default-schema-name specifies the name of the schema. See “Schemas” (page 165).

default-schema-name is an SQL identifier. For example, you can use MYSHEMA or myschema or a delimited identifier "My_Schema". See “Identifiers” (page 139).

Consideration 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.

Example of SET SCHEMA

Set the default schema name:

```sql
SET SCHEMA myschema;
```
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.

### 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 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.
TABLE Statement

- “Considerations for TABLE”
- “Example of TABLE”

The TABLE statement is equivalent to the query specification `SELECT * FROM table`.

```sql
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 74).

Example of TABLE

This TABLE statement returns the same result as `SELECT * FROM job`:

```sql
TABLE job;
```

<table>
<thead>
<tr>
<th>Job/Code</th>
<th>Job Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>MANAGER</td>
</tr>
<tr>
<td>200</td>
<td>PRODUCTION SUPV</td>
</tr>
<tr>
<td>250</td>
<td>ASSEMBLER</td>
</tr>
<tr>
<td>300</td>
<td>SALESREP</td>
</tr>
<tr>
<td>400</td>
<td>SYSTEM ANALYST</td>
</tr>
<tr>
<td>420</td>
<td>ENGINEER</td>
</tr>
<tr>
<td>450</td>
<td>PROGRAMMER</td>
</tr>
<tr>
<td>500</td>
<td>ACCOUNTANT</td>
</tr>
<tr>
<td>600</td>
<td>ADMINISTRATOR</td>
</tr>
<tr>
<td>900</td>
<td>SECRETARY</td>
</tr>
</tbody>
</table>

--- 10 row(s) selected.
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.

### 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 116).

**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 129).

**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 74).

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 166).

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 68).

[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 19).

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);
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]...

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 116).

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 103).

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.

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 the table.

See “SAMPLE Clause” (page 177).

**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 (\( 0 < \text{percent} \leq 100 \)). 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 \( period \). \( 0 < size \leq \text{period} \). 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 (\( \text{period} > 0 \)).

**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, reexecute the UPDATE STATISTICS statement for the table.
Authorization and Locking

To run the UPDATE STATISTICS statement against SQL tables, you must have the authority to read the user table for which statistics are generated. Because the histogram tables are registered in the schema of table, you must have the authority to read and write to this schema.

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:

<table>
<thead>
<tr>
<th>Single-Column Group</th>
<th>Single-Column Group Within Parentheses</th>
<th>Multicolumn Group</th>
</tr>
</thead>
<tbody>
<tr>
<td>ON CITY, STATE, ZIP</td>
<td>ON (CITY), (STATE), (ZIP)</td>
<td>ON (CITY, STATE) or ON (CITY, STATE, ZIP)</td>
</tr>
<tr>
<td>ON CITY TO ZIP</td>
<td>ON (CITY) TO (ZIP)</td>
<td></td>
</tr>
<tr>
<td>ON ZIP TO CITY</td>
<td>ON (ZIP) TO (CITY)</td>
<td></td>
</tr>
<tr>
<td>ON CITY, STATE TO ZIP</td>
<td>ON (CITY), (STATE) TO (ZIP)</td>
<td></td>
</tr>
<tr>
<td>ON CITY TO STATE, ZIP</td>
<td>ON (CITY) TO (STATE), (ZIP)</td>
<td></td>
</tr>
</tbody>
</table>

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.

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.
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.

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;
```
The UPSERT statement either updates a table if the row exists or inserts into a table if the row does not exist.

**Syntax Description of UPSERT**

```
UPSERT 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**

- **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 112).

  - 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 74).

- **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 129).

**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:

  ```sql
  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 74).

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 63).

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 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”
- “Schemas”
- “Search Condition”
- “Subquery”
- “Tables”
- “Views”
Authorization IDs

An authorization ID is a regular or delimited case-insensitive identifier that is used for an authorization operation. Authorization is the process of validating that a database user has permission to perform a specified SQL operation. An authorization ID can have a maximum of 128 characters.

All authorization IDs share the same namespace. An authorization ID can be a database user 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 114).

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:

<table>
<thead>
<tr>
<th>CHECK</th>
<th>Column or table constraint specifying a condition must be satisfied for each row in the table.</th>
</tr>
</thead>
<tbody>
<tr>
<td>FOREIGN KEY</td>
<td>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.</td>
</tr>
<tr>
<td>NOT NULL</td>
<td>Column constraint specifying the column cannot contain nulls.</td>
</tr>
<tr>
<td>PRIMARY KEY</td>
<td>Column or table constraint specifying the column or set of columns as the primary key for the table.</td>
</tr>
<tr>
<td>UNIQUE</td>
<td>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.</td>
</tr>
</tbody>
</table>

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 36) and “ALTER TABLE Statement” (page 26).
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. 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 74).

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.

“Indexes”
“Tables”
“Views”

Ownership

In Trafodion SQL, the creator of a schema owns all the objects defined in the schema and has all privileges on the objects defined in the schema.

For information on privileges on tables and views, see “CREATE TABLE Statement” (page 36) and “CREATE VIEW Statement” (page 47).
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 115).

Logical Names for SQL Objects

You may refer to an SQL table or view by using a one-part or two-part logical name, also called an ANSI name:

```
schema-name.object-name
```

In this two-part name, schema-name is the name of the schema, and object-name is the simple name of the table or view. Each of the parts is an SQL identifier. See “Identifiers” (page 139). Trafodion SQL automatically qualifies an object name with the schema name unless you explicitly specify schema names with the object name. 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 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
- Schema label
- Table value object (table or view)

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):

<table>
<thead>
<tr>
<th>Data Type</th>
<th>SQL Designation</th>
<th>Description</th>
<th>Size or Range (1)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Fixed-length character</strong></td>
<td>CHAR[ACTER]</td>
<td>Fixed-length character data</td>
<td>1 to 32707 characters (2)</td>
</tr>
<tr>
<td></td>
<td>NCHAR</td>
<td>Fixed-length character data in predefined national character set</td>
<td>1 to 32707 bytes (3) (7)</td>
</tr>
<tr>
<td></td>
<td>NATIONAL CHAR[ACTER]</td>
<td>Fixed-length character data in predefined national character set</td>
<td>1 to 32707 characters (3) (7)</td>
</tr>
<tr>
<td><strong>Variable-length character</strong></td>
<td>VARCHAR</td>
<td>Variable-length ASCII character string</td>
<td>1 to 32703 characters (4)</td>
</tr>
<tr>
<td></td>
<td>CHAR[ACTER] VARYING</td>
<td>Variable-length ASCII character string</td>
<td>1 to 32703 characters (4)</td>
</tr>
<tr>
<td></td>
<td>NCHAR VARYING</td>
<td>Variable-length ASCII character string</td>
<td>1 to 32703 bytes (4) (8)</td>
</tr>
<tr>
<td></td>
<td>NATIONAL CHAR[ACTER] VARYING</td>
<td>Variable-length ASCII character string</td>
<td>1 to 32703 characters (4) (8)</td>
</tr>
<tr>
<td><strong>Numeric</strong></td>
<td>NUMERIC (1, scale) to NUMERIC (128, scale)</td>
<td>Binary number with optional scale; signed or unsigned for 1 to 9 digits</td>
<td>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</td>
</tr>
<tr>
<td></td>
<td>SMALLINT</td>
<td>Binary integer; signed or unsigned</td>
<td>0 to 65535 unsigned, -32768 to +32767 signed; stored in 2 bytes</td>
</tr>
<tr>
<td></td>
<td>INTEGER</td>
<td>Binary integer; signed or unsigned</td>
<td>0 to 2,147,483,647 unsigned, -2,147,483,648 to +2,147,483,647 signed; stored in 4 bytes</td>
</tr>
<tr>
<td></td>
<td>LARGEINT</td>
<td>Binary integer; signed only</td>
<td>2<strong>63 to +(2</strong>63)-1; stored in 8 bytes</td>
</tr>
<tr>
<td><strong>Numeric (extended numeric precision)</strong></td>
<td>NUMERIC (precision 19 to 128)</td>
<td>Binary integer; signed or unsigned</td>
<td>Stored as multiple chunks of 16-bit integers, with a minimum storage length of 8 bytes.</td>
</tr>
<tr>
<td><strong>Floating point number</strong></td>
<td>FLOAT[(precision]]</td>
<td>Floating point number; precision designates from 1 through 52 bits of precision</td>
<td>+/- 2.2250738585072014e-308 through +/- 1.7976931348623157e+308; stored in 8 bytes</td>
</tr>
</tbody>
</table>

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 214).

The following table summarizes the Trafodion SQL data types:
<table>
<thead>
<tr>
<th>Type</th>
<th>SQL Designation</th>
<th>Description</th>
<th>Size or Range (1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>REAL</td>
<td>Floating point number (32 bits)</td>
<td>+/-.117549435e-38 through +/3.40282347e+38; stored in 4 bytes</td>
<td></td>
</tr>
<tr>
<td>DOUBLE PRECISION</td>
<td>Floating-point numbers (64 bits)</td>
<td>+/-2.2250738585072014e-308 through +/-1.7976931348623157e+308; stored in 8 bytes</td>
<td></td>
</tr>
<tr>
<td>Decimal number</td>
<td>DECIMAL (1, scale) to DECIMAL (18, scale)</td>
<td>Decimal number with optional scale; stored as ASCII characters; signed or unsigned for 1 to 9 digits; signed required for 10 or more digits</td>
<td>1 to 18 digits. Byte length equals the number of digits. Sign is stored as the first bit of the leftmost byte.</td>
</tr>
<tr>
<td>Date-Time</td>
<td>Point in time, using the Gregorian calendar and a 24 hour clock system. The five supported designations are listed below.</td>
<td></td>
<td>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</td>
</tr>
<tr>
<td>DATE</td>
<td>Date</td>
<td>Format as YYYY-MM-DD; actual database storage size is 4 bytes</td>
<td></td>
</tr>
<tr>
<td>TIME</td>
<td>Time of day, 24 hour clock, no time precision</td>
<td>Format as HH:MM:SS; actual database storage size is 3 bytes</td>
<td></td>
</tr>
<tr>
<td>TIME (with time precision)</td>
<td>Time of day, 24 hour clock, with time precision</td>
<td>Format as HH:MM:SS.FFFFFFF; actual database storage size is 7 bytes</td>
<td></td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>Point in time, no time precision</td>
<td>Format as YYYY-MM-DD HH:MM:SS; actual database storage size is 7 bytes</td>
<td></td>
</tr>
<tr>
<td>TIMESTAMP (with time precision)</td>
<td>Point in time, with time precision</td>
<td>Format as YYYY-MM-DD HH:MM:SS.FFFFFFF; actual database storage size is 11 bytes</td>
<td></td>
</tr>
<tr>
<td>Interval</td>
<td>INTERVAL</td>
<td>Duration of time; value is in the YEAR/MONTH range or the DAY/HOUR/MINUTE/SECOND/FRACTION range</td>
<td>YEAR no constraint(6) MONTH 0-11 DAY no constraint</td>
</tr>
</tbody>
</table>
### 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:

\[
\text{TIMESTAMP '2008-09-28 00:00:00' > TIMESTAMP '2008-06-26 00:00:00'}
\]

**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:

\[
\text{INTERVAL '02-01' YEAR TO MONTH > INTERVAL '00-01' YEAR TO MONTH}
\]

The field components of the INTERVAL do not have to be the same. For example, this predicate is also true:

\[
\text{INTERVAL '02-01' YEAR TO MONTH > INTERVAL '01' YEAR}
\]

**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. 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).

```sql
CREATE TABLE T(a NUMERIC(18,4), b NUMERIC(18,4));
INSERT INTO T VALUES (1.1234, 2.1234);
```
SELECT A+B FROM T;

---

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;

---

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;

---

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.
```
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:

<table>
<thead>
<tr>
<th>Character Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR</td>
<td>A column with fixed-length character data.</td>
</tr>
<tr>
<td>CHAR VARYING</td>
<td>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.</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>A column with varying-length character data. VARCHAR is equivalent to data type CHAR VARYING.</td>
</tr>
<tr>
<td>NCHAR</td>
<td>A column with data in the predefined national character set.</td>
</tr>
<tr>
<td>NCHAR VARYING</td>
<td>A column with varying-length data in the predefined national character set.</td>
</tr>
</tbody>
</table>

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.
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 214).

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:

```plaintext
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:

<table>
<thead>
<tr>
<th>yyyy</th>
<th>Year, from 0001 to 9999</th>
</tr>
</thead>
<tbody>
<tr>
<td>mm</td>
<td>Month, from 01 to 12</td>
</tr>
<tr>
<td>dd</td>
<td>Day, from 01 to 31</td>
</tr>
<tr>
<td>hh</td>
<td>Hour, from 00 to 23</td>
</tr>
<tr>
<td>mm</td>
<td>Minute, from 00 to 59</td>
</tr>
<tr>
<td>ss</td>
<td>Second, from 00 to 59</td>
</tr>
<tr>
<td>msssss</td>
<td>Microsecond, from 000000 to 999999</td>
</tr>
</tbody>
</table>

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)]

der-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 203), “DATE_ADD Function” (page 234), “DATE_SUB Function” (page 235), and “DATEADD Function” (page 236) 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:

\[
\text{max-leading-precision} = 18 - \text{fractional-precision} - 2 * (N - 1)
\]

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[GER] [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[GER] [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:

<table>
<thead>
<tr>
<th>Type of Expression</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Character Value Expressions” (page 129)</td>
<td>Operands can be combined with the concatenation operator (</td>
</tr>
<tr>
<td>“Datetime Value Expressions” (page 130)</td>
<td>Operands can be combined in specific ways with arithmetic operators. Example: <code>CURRENT_DATE + INTERVAL '1' DAY</code></td>
</tr>
<tr>
<td>“Interval Value Expressions” (page 133)</td>
<td>Operands can be combined in specific ways with addition and subtraction operators. Example: <code>INTERVAL '2' YEAR - INTERVAL '3' MONTH</code></td>
</tr>
<tr>
<td>“Numeric Value Expressions” (page 136)</td>
<td>Operands can be combined in specific ways with arithmetic operators. Example: <code>SALARY * 1.10</code></td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'ABILENE'</td>
<td>Character string literal.</td>
</tr>
<tr>
<td>'ABILENE'</td>
<td></td>
</tr>
<tr>
<td>'ABILENE'</td>
<td></td>
</tr>
<tr>
<td>'Customer '</td>
<td></td>
</tr>
<tr>
<td>CAST (order_date AS CHAR(10))</td>
<td>CAST function applied to a DATE value.</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Datetime Expression</th>
<th>Description</th>
<th>Result Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CURRENT_DATE + INTERVAL '1' DAY</td>
<td>The sum of the current date and an interval value of one day.</td>
<td>DATE</td>
</tr>
<tr>
<td>CURRENT_DATE + est_complete</td>
<td>The sum of the current date and the interval value in column EST_COMPLETE.</td>
<td>DATE</td>
</tr>
<tr>
<td>( SELECT ship_timestamp FROM project WHERE projcode=1000 ) + INTERVAL '07:04' DAY TO HOUR</td>
<td>The sum of the ship timestamp for the specified project and an interval value of seven days, four hours.</td>
<td>TIMESTAMP</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Operand 1</th>
<th>Operator</th>
<th>Operand 2</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Datetime</td>
<td>+ or –</td>
<td>Interval</td>
<td>Datetime</td>
</tr>
<tr>
<td>Interval</td>
<td>+</td>
<td>Datetime</td>
<td>Datetime</td>
</tr>
</tbody>
</table>

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 214). For more information on INTERVALS, see “Interval Value Expressions” (page 133)
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:

  ```sql
  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:

  ```sql
  DATE '2007-01-30' + INTERVAL '7' DAY + INTERVAL '1' MONTH
  ```

  You can avoid these unexpected results by using the “ADD_MONTHS Function” (page 203).

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:

```sql
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:

<table>
<thead>
<tr>
<th>PROJCODE</th>
<th>START_DATE</th>
<th>SHIP_TIMESTAMP</th>
<th>EST_COMPLETE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>2007-04-10</td>
<td>2007-04-21 08:15:00.00</td>
<td>15</td>
</tr>
<tr>
<td>945</td>
<td>2007-07-20</td>
<td>2007-07-21 08:15:00.00</td>
<td>30</td>
</tr>
<tr>
<td>920</td>
<td>2007-02-21</td>
<td>2007-03-12 09:45:00.00</td>
<td>20</td>
</tr>
<tr>
<td>134</td>
<td>2007-11-20</td>
<td>2008-01-01 00:00:00.00</td>
<td>30</td>
</tr>
</tbody>
</table>

- Add an interval value qualified by YEAR to a datetime value:

  ```sql
  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:

  ```sql
  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:

  ```sql
  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 130).

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:

<table>
<thead>
<tr>
<th>Datetime Expression</th>
<th>Description</th>
<th>Result Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>CURRENT_DATE - start_date</code></td>
<td>By default, the interval difference between the current date and the value in column <code>START_DATE</code> is expressed in days. You are not required to specify the interval qualifier.</td>
<td>INTERVAL DAY (12)</td>
</tr>
<tr>
<td><code>INTERVAL '3' DAY - INTERVAL '2' DAY</code></td>
<td>The difference of two interval literals. The result is 1 day.</td>
<td>INTERVAL DAY (3)</td>
</tr>
<tr>
<td><code>INTERVAL '3' DAY + INTERVAL '2' DAY</code></td>
<td>The sum of two interval literals. The result is 5 days.</td>
<td>INTERVAL DAY (3)</td>
</tr>
<tr>
<td><code>INTERVAL '2' YEAR - INTERVAL '3' MONTH</code></td>
<td>The difference of two interval literals. The result is 1 year, 9 months.</td>
<td>INTERVAL YEAR (3) TO MONTH</td>
</tr>
</tbody>
</table>

**Restrictions on Operations**
You can use datetime and interval operands with arithmetic operators in an interval value expression only in these combinations:

<table>
<thead>
<tr>
<th>Operand 1</th>
<th>Operator</th>
<th>Operand 2</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Datetime</td>
<td>–</td>
<td>Datetime</td>
<td>Interval</td>
</tr>
<tr>
<td>Interval</td>
<td>+ or –</td>
<td>Interval</td>
<td>Interval</td>
</tr>
<tr>
<td>Interval</td>
<td>* or /</td>
<td>Numeric</td>
<td>Interval</td>
</tr>
<tr>
<td>Numeric</td>
<td>*</td>
<td>Interval</td>
<td>Interval</td>
</tr>
</tbody>
</table>

This table lists valid combinations of datetime and interval arithmetic operators, and the data type of the result:

<table>
<thead>
<tr>
<th>Operands</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date + Interval or Interval + Date</td>
<td>Date</td>
</tr>
<tr>
<td>Date + Numeric or Numeric + Date</td>
<td>Date</td>
</tr>
<tr>
<td>Date * Numeric</td>
<td>Date</td>
</tr>
<tr>
<td>Date – Interval</td>
<td>Date</td>
</tr>
<tr>
<td>Date – Date</td>
<td>Interval</td>
</tr>
<tr>
<td>Time + Interval or Interval + Time</td>
<td>Time</td>
</tr>
<tr>
<td>Time + Numeric or Numeric + Time</td>
<td>Time</td>
</tr>
<tr>
<td>Time * Number</td>
<td>Time</td>
</tr>
<tr>
<td>Time – Interval</td>
<td>Time</td>
</tr>
<tr>
<td>Timestamp + Interval or Interval + Timestamp</td>
<td>Timestamp</td>
</tr>
<tr>
<td>Timestamp + Numeric or Numeric + Timestamp</td>
<td>Timestamp</td>
</tr>
<tr>
<td>Timestamp * Numeric</td>
<td>Timestamp</td>
</tr>
<tr>
<td>Timestamp – Interval</td>
<td>Timestamp</td>
</tr>
<tr>
<td>year-month Interval + year-month Interval</td>
<td>year-month Interval</td>
</tr>
<tr>
<td>day-time Interval + day-time Interval</td>
<td>day-time Interval</td>
</tr>
<tr>
<td>year-month Interval – year-month Interval</td>
<td>year-month Interval</td>
</tr>
<tr>
<td>day-time Interval – day-time Interval</td>
<td>day-time Interval</td>
</tr>
<tr>
<td>Time – Time</td>
<td>Interval</td>
</tr>
<tr>
<td>Timestamp – Timestamp</td>
<td>Interval</td>
</tr>
<tr>
<td>Interval * Number or Number * Interval</td>
<td>Interval</td>
</tr>
<tr>
<td>Interval / Number</td>
<td>Interval</td>
</tr>
<tr>
<td>Interval – Interval or Interval + Interval</td>
<td>Interval</td>
</tr>
</tbody>
</table>

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:
  
  ```sql
  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:
  
  ```sql
  (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:

```sql
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:

```
<table>
<thead>
<tr>
<th>PROJCODE</th>
<th>START_DATE</th>
<th>SHIP_TIMESTAMP</th>
<th>EST_COMPLETE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>2007-04-10</td>
<td>2007-04-21:08:15:00.0000</td>
<td>15</td>
</tr>
<tr>
<td>2000</td>
<td>2007-06-10</td>
<td>2007-07-21:08:30:00.0000</td>
<td>30</td>
</tr>
<tr>
<td>2500</td>
<td>2007-10-10</td>
<td>2007-12-21:09:00.0000</td>
<td>60</td>
</tr>
<tr>
<td>3000</td>
<td>2007-08-21</td>
<td>2007-10-21:08:10.0000</td>
<td>60</td>
</tr>
<tr>
<td>4000</td>
<td>2007-09-21</td>
<td>2007-10-21:10:15:00.0000</td>
<td>30</td>
</tr>
</tbody>
</table>
```

• 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:

```sql
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 03:08:57.087086
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.
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:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-57</td>
<td>Numeric literal.</td>
</tr>
<tr>
<td>salary * 1.10</td>
<td>The product of the values in the SALARY column and a numeric literal.</td>
</tr>
<tr>
<td>unit_price * qty_ordered</td>
<td>The product of the values in the UNIT_PRICE and QTY.ORDERED columns.</td>
</tr>
<tr>
<td>12 * (7 - 4)</td>
<td>An expression whose operands are numeric literals.</td>
</tr>
<tr>
<td>COUNT (DISTINCT city)</td>
<td>Function applied to the values in a column.</td>
</tr>
</tbody>
</table>
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 (" "). An identifier of either type can contain up to 128 characters.

Regular Identifiers

Regular identifiers begin with a letter (A through Z or 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.

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 34).
Trafodion SQL supports these types of keys:

- "Clustering Keys"
- "SYSKEY" (page 141)
- "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 34).

**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:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Character String Literals”</td>
<td>A series of characters enclosed in single quotes. Example: ‘Planning’</td>
</tr>
<tr>
<td>“Datetime Literals”</td>
<td>Begins with keyword DATE, TIME, or TIMESTAMP and followed by a character string. Example: DATE ‘1990-01-22’</td>
</tr>
<tr>
<td>“Interval Literals”</td>
<td>Begins with keyword INTERVAL and followed by a character string and an interval qualifier. Example: INTERVAL ‘2-7’ YEAR TO MONTH</td>
</tr>
<tr>
<td>“Numeric Literals”</td>
<td>A simple numeric literal (one without an exponent) or a numeric literal in scientific notation. Example: 99E-2</td>
</tr>
</tbody>
</table>

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.
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 122). 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.

<table>
<thead>
<tr>
<th>Character String Data Type</th>
<th>Character String Literal Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR (12) UPSHIFT</td>
<td>'PLANNING'</td>
</tr>
<tr>
<td>VARCHAR (18)</td>
<td>'NEW YORK'</td>
</tr>
</tbody>
</table>

- 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.msssss Default
- hh:mm:ss.msssss [am | pm] USA
- hh.mm.ss.msssss European

timestamp is:

- yyyy-mm-dd hh:mm:ss.msssss Default
- mm/dd/yyyy hh:mm:ss.msssss [am | pm] USA
- dd.mm.yyyy hh.mm.ss.msssss European

date, time, timestamp

specify the datetime literal strings whose component fields are:

<table>
<thead>
<tr>
<th>yyyy</th>
<th>Year, from 0001 to 9999</th>
</tr>
</thead>
<tbody>
<tr>
<td>mm</td>
<td>Month, from 01 to 12</td>
</tr>
<tr>
<td>dd</td>
<td>Day, from 01 to 31</td>
</tr>
<tr>
<td>hh</td>
<td>Hour, from 00 to 23</td>
</tr>
<tr>
<td>mm</td>
<td>Minute, from 00 to 59</td>
</tr>
<tr>
<td>ss</td>
<td>Second, from 00 to 59</td>
</tr>
<tr>
<td>msssss</td>
<td>Microsecond, from 000000 to 999999</td>
</tr>
<tr>
<td>am</td>
<td>AM or am, indicating time from midnight to before noon</td>
</tr>
<tr>
<td>pm</td>
<td>PM or pm, indicating time from noon to before midnight</td>
</tr>
</tbody>
</table>

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).

\[-\text{INTERVAL} \ [-\{\text{’year-month’} \mid \text{’day:time’}\} \text{ interval-qualifier}\]

\text{interval-qualifier is:}
\text{start-field TO end-field} \mid \text{single-field}

\text{start-field is:}
\{\text{YEAR} \mid \text{MONTH} \mid \text{DAY} \mid \text{HOUR} \mid \text{MINUTE}\} [(\text{leading-precision})]

\text{end-field is:}
\text{YEAR} \mid \text{MONTH} \mid \text{DAY} \mid \text{HOUR} \mid \text{MINUTE} \mid \text{SECOND} [(\text{fractional-precision})]

\text{single-field is:}
\text{start-field} \mid \text{SECOND} [(\text{leading-precision}, \text{fractional-precision})]

\text{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.

\{\text{YEAR} \mid \text{MONTH} \mid \text{DAY} \mid \text{HOUR} \mid \text{MINUTE}\} [(\text{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.

\text{YEAR} \mid \text{MONTH} \mid \text{DAY} \mid \text{HOUR} \mid \text{MINUTE} \mid \text{SECOND} [(\text{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.

\text{start-field} \mid \text{SECOND} [(\text{leading-precision}, \text{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 125) and “Interval Value Expressions” (page 133).
'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:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>years</td>
<td>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.</td>
</tr>
<tr>
<td>months</td>
<td>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.</td>
</tr>
<tr>
<td>days</td>
<td>Unsigned integer that specifies a 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).</td>
</tr>
<tr>
<td>hours</td>
<td>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.</td>
</tr>
<tr>
<td>minutes</td>
<td>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.</td>
</tr>
<tr>
<td>seconds</td>
<td>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.</td>
</tr>
<tr>
<td>fraction</td>
<td>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.</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Interval Literal</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTERVAL '1' MONTH</td>
<td>Interval of 1 month</td>
</tr>
<tr>
<td>INTERVAL '7' DAY</td>
<td>Interval of 7 days</td>
</tr>
<tr>
<td>INTERVAL '2-7' YEAR TO MONTH</td>
<td>Interval of 2 years, 7 months</td>
</tr>
<tr>
<td>INTERVAL '5:2:15:36.33' DAY TO SECOND(2)</td>
<td>Interval of 5 days, 2 hours, 15 minutes, and 36.33 seconds</td>
</tr>
<tr>
<td>INTERVAL - '5' DAY</td>
<td>Interval that subtracts 5 days</td>
</tr>
<tr>
<td>INTERVAL '100' DAY(3)</td>
<td>Interval of 100 days. This example requires an explicit leading precision of 3 because the default is 2.</td>
</tr>
<tr>
<td>INTERVAL '364 23' DAY(3) TO HOUR</td>
<td>Interval of 364 days, 23 hours. The separator for the day and hour fields can be a space or a colon.</td>
</tr>
</tbody>
</table>

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:

\[ [+|-] \text{unsigned-integer}\[.\text{unsigned-integer}][\text{exact-numeric-literal}][+|-]\text{unsigned-integer} \]

**approximate-numeric-literal** is:

\[ \text{mantissa}\{E|e\}\text{exponent} \]

**mantissa** is:

\[ \text{exact-numeric-literal} \]

**exponent** is:

\[ [+|-] \text{unsigned-integer} \]

**unsigned-integer** is:

\[ \text{digit}... \]

**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:

```sql
INSERT INTO XYZ VALUES (1.00000000E-10)
```

has more precision than:

```sql
INSERT INTO XYZ VALUES (1.0E-10)
```
Examples of Numeric Literals

These are all numeric literals, along with their display format:

<table>
<thead>
<tr>
<th>Literal</th>
<th>Display Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>477</td>
<td>477</td>
</tr>
<tr>
<td>580.45</td>
<td>580.45</td>
</tr>
<tr>
<td>+005</td>
<td>5</td>
</tr>
<tr>
<td>-.3175</td>
<td>-.3175</td>
</tr>
<tr>
<td>1300000000</td>
<td>1300000000</td>
</tr>
<tr>
<td>99.</td>
<td>99</td>
</tr>
<tr>
<td>-0.123456789012345678</td>
<td>-.123456789012345678</td>
</tr>
<tr>
<td>99E-2</td>
<td>9.9000000E-001</td>
</tr>
<tr>
<td>12.3e+5</td>
<td>1.22999999E+06</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>Clause</th>
<th>Consideration</th>
</tr>
</thead>
<tbody>
<tr>
<td>DISTINCT</td>
<td>Nulls are considered duplicates; therefore, a result has at most one null.</td>
</tr>
<tr>
<td>GROUP BY</td>
<td>The result has at most one null group.</td>
</tr>
<tr>
<td>ORDER BY</td>
<td>Nulls are considered greater than nonnull values.</td>
</tr>
</tbody>
</table>
### Null and Expression Evaluation Comparison

<table>
<thead>
<tr>
<th>Expression Type</th>
<th>Condition</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean operators (AND, OR, NOT)</td>
<td>Either operand is null.</td>
<td>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.</td>
</tr>
<tr>
<td>Arithmetic operators</td>
<td>Either or both operands are null.</td>
<td>The result is null.</td>
</tr>
<tr>
<td>NULL predicate</td>
<td>The operand is null.</td>
<td>The result is true.</td>
</tr>
<tr>
<td>Aggregate (or set) functions (except COUNT)</td>
<td>Some rows have null columns. The function is evaluated after eliminating nulls.</td>
<td>The result is null if set is empty.</td>
</tr>
<tr>
<td>COUNT(*)</td>
<td>The function does not eliminate nulls.</td>
<td>The result is the number of rows in the table whether or not the rows are null.</td>
</tr>
<tr>
<td>COUNT COUNT DISTINCT</td>
<td>The function is evaluated after eliminating nulls.</td>
<td>The result is zero if set is empty.</td>
</tr>
<tr>
<td>Comparison: =, &lt;, &gt;, &lt;=, &gt;, =, LIKE</td>
<td>Either operand is null.</td>
<td>The result is null.</td>
</tr>
<tr>
<td>IN predicate</td>
<td>Some expressions in the IN value list are null.</td>
<td>The result is null if all of the expressions are null.</td>
</tr>
<tr>
<td>Subquery</td>
<td>No rows are returned.</td>
<td>The result is null.</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Predicate Description</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>BETWEEN Predicate</td>
<td>Determines whether a sequence of values is within a range of sequences of values.</td>
</tr>
<tr>
<td>Comparison Predicates</td>
<td>Compares the values of sequences of expressions, or compares the values of sequences of row values that are the result of row subqueries.</td>
</tr>
<tr>
<td>EXISTS Predicate</td>
<td>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.</td>
</tr>
<tr>
<td>IN Predicate</td>
<td>Determines if a sequence of values is equal to any of the sequences of values in a list of sequences.</td>
</tr>
<tr>
<td>LIKE Predicate</td>
<td>Searches for character strings that match a pattern.</td>
</tr>
<tr>
<td>NULL Predicate</td>
<td>Determines whether all the values in a sequence of values are null.</td>
</tr>
<tr>
<td>Quantified Comparison Predicates</td>
<td>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.</td>
</tr>
</tbody>
</table>

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 129).
  - `row-subquery` is a subquery that returns a single row (consisting of a sequence of values). See “Subquery” (page 168).

The three `row-value-constructor` 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-constructor` must be comparable. Respective values are values with the same ordinal position in the two lists. See “Comparable and Compatible Data Types” (page 119).
Considerations for BETWEEN

**Logical Equivalents Using AND and OR**

The predicate \( expr_1 \) BETWEEN \( expr_2 \) AND \( expr_3 \) is true if and only if this condition is true:

\[
expr_2 \leq expr_1 \text{ AND } expr_1 \leq expr_3
\]

The predicate \( expr_1 \) NOT BETWEEN \( expr_2 \) AND \( expr_3 \) is true if and only if this condition is true:

\[
expr_2 > expr_1 \text{ OR } expr_1 > expr_3
\]

**Descending Columns in Keys**

If a clause specifies a column in a key BETWEEN \( expr_2 \) and \( expr_3 \), \( expr_3 \) must be greater than \( expr_2 \) 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:

  \[
  \text{qty\_on\_hand \times price} \text{ BETWEEN 1000.00 AND 10000.00}
  \]

- This predicate is true if the part cost is less than $5 or more than $800:

  \[
  \text{partcost NOT BETWEEN 5.00 AND 800.00}
  \]

- This BETWEEN predicate selects the part number 6400:

  ```sql
  SELECT * FROM partsupp
  WHERE partnum BETWEEN 6400 AND 6700
  AND partcost > 300.00;
  ```

<table>
<thead>
<tr>
<th>Part/Num</th>
<th>Supp/Num</th>
<th>Part/Cost</th>
<th>Qty/Rec</th>
</tr>
</thead>
<tbody>
<tr>
<td>6400</td>
<td>1</td>
<td>390.00</td>
<td>50</td>
</tr>
<tr>
<td>6401</td>
<td>2</td>
<td>500.00</td>
<td>20</td>
</tr>
<tr>
<td>6401</td>
<td>3</td>
<td>480.00</td>
<td>38</td>
</tr>
</tbody>
</table>

  --- 3 row(s) selected.

- Find names between Jody Selby and Gene Wright:

  ```sql
  (last\_name, first\_name) BETWEEN ('SELBY', 'JODY') AND ('WRIGHT', 'GENE')
  ```

  The name Barbara Swift would meet the criteria; the name Mike Wright would not.

  ```sql
  SELECT empnum, first\_name, last\_name
  FROM persnl.employee
  WHERE (last\_name, first\_name) BETWEEN ('SELBY', 'JODY') AND ('WRIGHT', 'GENE');
  ```

<table>
<thead>
<tr>
<th>EMPNUM</th>
<th>FIRST_NAME</th>
<th>LAST_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>43</td>
<td>PAUL</td>
<td>WINTER</td>
</tr>
<tr>
<td>72</td>
<td>GLENN</td>
<td>THOMAS</td>
</tr>
<tr>
<td>74</td>
<td>JOHN</td>
<td>WALKER</td>
</tr>
</tbody>
</table>

  ...  

  --- 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.

\[
\text{row-value-constructor comparison-op row-value-constructor}
\]

\[
\begin{align*}
\text{comparison-op is:} & \\
\text{=} & \text{ Equal} \\
\text{<>} & \text{ Not equal} \\
\text{<} & \text{ Less than} \\
\text{>} & \text{ Greater than} \\
\text{<=} & \text{ Less than or equal to} \\
\text{>=} & \text{ Greater than or equal to}
\end{align*}
\]

\[
\text{row-value-constructor is:} \\
(\text{expression} [, \text{expression}] ...)
\]

| row-subquery |

\[
\text{row-value-constructor}
\]

specifies an operand of a comparison predicate. The two operands can be either of these:

- \((\text{expression} [, \text{expression}] ...))\)
  - is a sequence of SQL value expressions, separated by commas and enclosed in parentheses. \text{expression} cannot include an aggregate function unless expression is in a HAVING clause. \text{expression} can be a scalar subquery (a subquery that returns a single row consisting of a single column). See “Expressions” (page 129).
- \text{row-subquery}
  - is a subquery that returns a single row (consisting of a sequence of values). See “Subquery” (page 168).

The two \text{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 \text{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 119).

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:

\[
X=(X_1, X_2, \ldots, X_n), \quad Y=(Y_1, Y_2, \ldots, Y_n).
\]

Predicate \(X=Y\) is true if for all \(i=1, \ldots, n\): \(X_i=Y_i\). For this predicate, Trafodion SQL must look through all values. Predicate \(X = Y\) is false if for some \(i\) \(X_i<>Y_i\). 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 \(X_i<>Y_1\), 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 \(i=1, \ldots, n\): \(X_i=Y_i\). In this situation, Trafodion SQL must look through all components.

Predicate \(X>Y\) is true if for some index \(m\) \(X_m>Y_m\) and for all \(i=1, \ldots, m-1\): \(X_i=Y_i\). Trafodion SQL does not look through all components. It stops when it finds the first nonequal components, \(X_m<>Y_m\). If \(X_m>Y_m\), the predicate is true. Otherwise the predicate is false. The predicate is also false if all components are equal, or \(X=Y\).
Predicate $X \geq Y$ 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, $X_m \not= Y_m$. If $X_m > Y_m$, the predicate is true. Otherwise, it is false.

Predicate $X < Y$ is true if for some index $m$ $X_m < Y_m$, and for all $i = 1, \ldots, m-1$: $X_i = Y_i$. Trafodion SQL does not look through all components. It stops when it finds the first nonequal components $X_m > Y_m$. If $X_m < Y_m$, the predicate is true. Otherwise, the predicate is false. The predicate is also false if all components are equal, or $X = Y$.

Predicate $X \leq Y$ 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, $X_m \not= Y_m$. If $X_m < Y_m$, 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:
  
  $$\text{custnum} = 3210$$

- This predicate is true if the salary is greater than the average salary of all employees:
  
  $$\text{salary} >$$
  
  $$(\text{SELECT AVG (salary) FROM persnl.employee});$$

- This predicate is true if the customer name is BACIGALUPI:
  
  $$\text{custname} = 'BACIGALUPI'$$

- This predicate evaluates to unknown for any rows in either CUSTOMER or ORDERS that contain null in the CUSTNUM column:
  
  $$\text{customer.custnum} > \text{orders.custnum}$$

- 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:
  
  $$(\text{last\_name, first\_name} > ('MOSS', 'DUNCAN'))$$

  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 \( \text{START\_DATE} \) and the result of the \text{CURRENT\_DATE} function:
  \[ \text{START\_DATE} < \text{CURRENT\_DATE} \]

- Compare two datetime values \( \text{START\_DATE} \) and \( \text{SHIP\_TIMESTAMP} \):
  \[ \text{CAST} (\text{start\_date AS TIMESTAMP}) < \text{ship\_timestamp} \]

- Compare two \text{INTERVAL} values:
  \[ \text{JOB1\_TIME} < \text{JOB2\_TIME} \]

  Suppose that \( \text{JOB1\_TIME} \), defined as \text{INTERVAL DAY TO MINUTE}, is 2 days 3 hours, and
  \( \text{JOB2\_TIME} \), defined as \text{INTERVAL DAY TO HOUR}, is 3 days.
  To evaluate the predicate, Trafodion SQL converts the two \text{INTERVAL} values to \text{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 \text{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:

\[
\begin{align*}
\text{SELECT} & \quad \text{first\_name, last\_name, deptnum, salary} \\
& \quad \text{FROM persnl.employee} \\
& \quad \text{WHERE salary} > (\text{SELECT} \quad \text{MAX} (\text{salary}) \\
& \quad \quad \text{FROM persnl.employee} \\
& \quad \quad \text{WHERE deptnum} = 1500); \\
\end{align*}
\]

<table>
<thead>
<tr>
<th>FIRST_NAME</th>
<th>LAST_NAME</th>
<th>DEPTNUM</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROGER</td>
<td>GREEN</td>
<td>9000</td>
<td>175500.00</td>
</tr>
<tr>
<td>KATHRYN</td>
<td>HALL</td>
<td>4000</td>
<td>96000.00</td>
</tr>
<tr>
<td>RACHEL</td>
<td>MCKAY</td>
<td>4000</td>
<td>118000.00</td>
</tr>
<tr>
<td>THOMAS</td>
<td>RUDLOFF</td>
<td>2000</td>
<td>138000.40</td>
</tr>
<tr>
<td>JANE</td>
<td>RAYMOND</td>
<td>3000</td>
<td>136000.00</td>
</tr>
<tr>
<td>JERRY</td>
<td>HOWARD</td>
<td>1000</td>
<td>137000.10</td>
</tr>
</tbody>
</table>

--- 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:

\[
\begin{align*}
\text{SELECT} & \quad \text{first\_name, last\_name, deptnum, salary} \\
& \quad \text{FROM persnl.employee} \\
& \quad \text{WHERE deptnum} <> 1500 \text{ AND} \\
& \quad \text{salary} < (\text{SELECT} \quad \text{MIN} (\text{salary}) \\
& \quad \quad \text{FROM persnl.employee} \\
& \quad \quad \text{WHERE deptnum} <> 1500 \text{ AND} \\
& \quad \quad \text{salary} > (\text{SELECT} \quad \text{AVG} (\text{salary}) \\
& \quad \quad \quad \text{FROM persnl.employee} \\
& \quad \quad \quad \text{WHERE deptnum} = 1500)); \\
\end{align*}
\]

<table>
<thead>
<tr>
<th>FIRST_NAME</th>
<th>LAST_NAME</th>
<th>DEPTNUM</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>JESSICA</td>
<td>CRINER</td>
<td>3500</td>
<td>39500.00</td>
</tr>
<tr>
<td>ALAN</td>
<td>TERRY</td>
<td>3000</td>
<td>39500.00</td>
</tr>
<tr>
<td>DINAH</td>
<td>CLARK</td>
<td>9000</td>
<td>37000.00</td>
</tr>
</tbody>
</table>
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.

\[
[\text{NOT}] \text{ EXISTS subquery}
\]

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 168).

**Examples of EXISTS**

- **Find locations of employees with job code 300:**

  ```sql
  SELECT deptnum, location FROM persnl.dept D
  WHERE EXISTS
    (SELECT jobcode FROM persnl.employee E
     WHERE D.deptnum = E.deptnum AND jobcode = 300);
  ```

<table>
<thead>
<tr>
<th>DEPTNUM</th>
<th>LOCATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>3000</td>
<td>NEW YORK</td>
</tr>
<tr>
<td>3100</td>
<td>TORONTO</td>
</tr>
<tr>
<td>3200</td>
<td>FRANKFURT</td>
</tr>
<tr>
<td>3300</td>
<td>LONDON</td>
</tr>
<tr>
<td>3500</td>
<td>HONG KONG</td>
</tr>
</tbody>
</table>

  --- 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:**

  ```sql
  SELECT deptname FROM persnl.dept D
  WHERE NOT EXISTS
    (SELECT jobcode FROM persnl.employee E
     WHERE D.deptnum = E.deptnum AND jobcode = 420);
  ```

<table>
<thead>
<tr>
<th>DEPTNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>FINANCE</td>
</tr>
<tr>
<td>PERSONNEL</td>
</tr>
<tr>
<td>INVENTORY</td>
</tr>
</tbody>
</table>

  --- 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);

<table>
<thead>
<tr>
<th>PARTNUM</th>
<th>SUPPNUM</th>
</tr>
</thead>
<tbody>
<tr>
<td>212</td>
<td>1</td>
</tr>
<tr>
<td>212</td>
<td>3</td>
</tr>
<tr>
<td>2001</td>
<td>1</td>
</tr>
<tr>
<td>2003</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

--- 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.

\[
\text{row-value-constructor} = \begin{cases} 
\text{[NOT] IN \{table-subquery | in-value-list\}} \\
\text{row-subquery} \\
\end{cases}
\]

\[
\text{in-value-list} = \begin{cases} 
\text{\{expression [,expression]...\}} \\
\text{row-subquery} \\
\end{cases}
\]

\[
\text{row-value-constructor} \text{ specifies the first operand of the IN predicate. The first operand can be either of:}
\]

\[
\text{(expression [,expression]...)} \\
\text{row-subquery}
\]

\[
\text{in-value-list} \text{ is:}
\]

\[
\text{(expression [,expression]...)} \\
\]

\[
\text{row-value-constructor}
\]

\[
\text{is a sequence of SQL value expressions, separated by commas and enclosed in parentheses. expression cannot include an aggregate function unless expression is in a HAVING clause. expression can be a scalar subquery (a subquery that returns a single row consisting of a single column). See “Expressions” (page 129).}
\]

\[
\text{row-subquery}
\]

\[
\text{is a subquery that returns a single row (consisting of a sequence of values). See “Subquery” (page 168).}
\]

\[
\text{table-subquery}
\]

\[
\text{is a subquery that returns a table (consisting of rows of columns). The table specifies rows of values to be compared with the row of values specified by the row-value-constructor. The number of values of the row-value-constructor must be equal to the number of columns in the result table of the table-subquery, and the data types of the values must be comparable.}
\]

\[
\text{in-value-list}
\]

\[
\text{is a sequence of SQL value expressions, separated by commas and enclosed in parentheses. expression cannot include an aggregate function defined on a column. expression can be a scalar subquery (a subquery that returns a single row consisting of a single column). In this case, the result of the row-value-constructor is a single value. The data types of the values must be comparable. The number of expressions in the in-value-list can have at least 5000 expressions.}
\]
Considerations for IN

Logical Equivalent Using ANY (or SOME)
The predicate \( expr \ IN (expr1, expr2, \ldots) \) is true if and only if the following predicate is true:
\[ expr = \text{ANY} (expr1, expr2, \ldots) \]

**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 74).

- 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:

  ```sql
  SELECT last_name, first_name, empnum
  FROM persnl.employee
  WHERE empnum IN (39, 337, 452);
  ```

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>FIRST_NAME</th>
<th>EMPNUM</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLARK</td>
<td>DINAH</td>
<td>337</td>
</tr>
<tr>
<td>SAPPERT</td>
<td>KLAUS</td>
<td>39</td>
</tr>
</tbody>
</table>
  --- 2 row(s) selected.

- Find those items in PARTS whose part number is not in the PARTLOC table:

  ```sql
  SELECT partnum, partdesc
  FROM sales.parts
  WHERE partnum NOT IN
    (SELECT partnum
     FROM sales.partloc)
  ```
FROM invent.partloc);

<table>
<thead>
<tr>
<th>PARTNUM</th>
<th>PARTDESC</th>
</tr>
</thead>
<tbody>
<tr>
<td>186</td>
<td>186 MegaByte Disk</td>
</tr>
</tbody>
</table>

--- 1 row(s) selected.

- Find those items (and their suppliers) in PARTS that have a supplier in the PARTSUPP table:

  ```sql
  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:

  ```sql
  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 160) and “Escape Characters” (page 160).

See “Character Value Expressions” (page 129).

**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 160)
- “Underscore (_)” (page 160)

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 a 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 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\%’.

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\%’.

160 SQL Language Elements
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 149).

```
row-value-constructor IS [NOT] NULL
```

```
row-value-constructor is:
  (expression [,expression]...) | row-subquery
```

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 129).

```
row-subquery
```

is a subquery that returns a single row (consisting of a sequence of values). See “Subquery” (page 168).

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`.

<table>
<thead>
<tr>
<th>Expressions</th>
<th><code>rvc</code> IS NULL</th>
<th><code>rvc</code> IS NOT NULL</th>
<th>NOT <code>rvc</code> IS NULL</th>
<th>NOT <code>rvc</code> IS NOT NULL</th>
</tr>
</thead>
<tbody>
<tr>
<td>degree 1: null</td>
<td>TRUE</td>
<td>FALSE</td>
<td>FALSE</td>
<td>TRUE</td>
</tr>
<tr>
<td>degree 1: not null</td>
<td>FALSE</td>
<td>TRUE</td>
<td>TRUE</td>
<td>FALSE</td>
</tr>
<tr>
<td>degree&gt;1: all null</td>
<td>TRUE</td>
<td>FALSE</td>
<td>FALSE</td>
<td>TRUE</td>
</tr>
<tr>
<td>degree&gt;1: some null</td>
<td>FALSE</td>
<td>FALSE</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
<tr>
<td>degree&gt;1: none null</td>
<td>FALSE</td>
<td>TRUE</td>
<td>TRUE</td>
<td>FALSE</td>
</tr>
</tbody>
</table>

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

determines 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 129).

row-subquery

is a subquery that returns a single row (consisting of a sequence of values). See “Subquery” (page 168).

quantifier is:
  ALL | ANY | SOME

A quantified comparison predicate compares the values of sequences of expressions to the values in each row selected by the 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.

A quantified comparison predicate compares the values of sequences of expressions to the values in each row selected by the 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.

A quantified comparison predicate compares the values of sequences of expressions to the values in each row selected by the 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.

A quantified comparison predicate compares the values of sequences of expressions to the values in each row selected by the 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.

A quantified comparison predicate compares the values of sequences of expressions to the values in each row selected by the 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.

A quantified comparison predicate compares the values of sequences of expressions to the values in each row selected by the 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.

A quantified comparison predicate compares the values of sequences of expressions to the values in each row selected by the 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.

A quantified comparison predicate compares the values of sequences of expressions to the values in each row selected by the 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.

A quantified comparison predicate compares the values of sequences of expressions to the values in each row selected by the 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.
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 $R$\(\,\text{comparison-op}\,\,\text{ALL}\,\,T\)**

If $T$ is empty or if $R$ \(\text{comparison-op}\,RT\) is true for every row $RT$ in $T$, the \(\text{comparison-op}\,\text{ALL}\) predicate is true.

If $R$ \(\text{comparison-op}\,RT\) is false for at least one row $RT$ in $T$, the \(\text{comparison-op}\,\text{ALL}\) predicate is false.

**Result of $R$\(\,\text{comparison-op}\,\text{ANY}\,T\) or $R$\(\,\text{comparison-op}\,\text{SOME}\,T\)**

If $T$ is empty or if $R$ \(\text{comparison-op}\,RT\) is false for every row $RT$ in $T$, the \(\text{comparison-op}\,\text{ANY}\) predicate is false.

If $R$ \(\text{comparison-op}\,RT\) is true for at least one row $RT$ in $T$, the \(\text{comparison-op}\,\text{ANY}\) 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:
  
  ```sql
  salary > ALL (SELECT salary 
  FROM persnl.employee 
  WHERE jobcode = 420)
  ```

  Consider this SELECT statement using the preceding predicate:

  ```sql
  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:

  ```sql
  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:

  ```sql
  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`
This predicate is true if the part number is equal to any part number with more than five units in stock:

```sql
partnum = ANY (SELECT partnum
FROM sales.odetail
WHERE qty_ordered > 5)
```

Consider this SELECT statement using the preceding predicate:

```sql
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:

```sql
SELECT partnum
FROM sales.odetail
WHERE qty_ordered > 5;
```

<table>
<thead>
<tr>
<th>Part/Num</th>
</tr>
</thead>
<tbody>
<tr>
<td>2403</td>
</tr>
<tr>
<td>5100</td>
</tr>
<tr>
<td>5103</td>
</tr>
<tr>
<td>6301</td>
</tr>
<tr>
<td>6500</td>
</tr>
</tbody>
</table>

--- 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:

```sql
SELECT ordernum, partnum, qty_ordered
FROM sales.odetail
WHERE partnum = ANY (SELECT partnum
FROM sales.odetail
WHERE qty_ordered > 5);
```

<table>
<thead>
<tr>
<th>Order/Num</th>
<th>Part/Num</th>
<th>Qty/Ord</th>
</tr>
</thead>
<tbody>
<tr>
<td>100210</td>
<td>244</td>
<td>3</td>
</tr>
<tr>
<td>100210</td>
<td>2001</td>
<td>3</td>
</tr>
<tr>
<td>100210</td>
<td>2403</td>
<td>6</td>
</tr>
<tr>
<td>100210</td>
<td>5100</td>
<td>10</td>
</tr>
<tr>
<td>100250</td>
<td>244</td>
<td>4</td>
</tr>
<tr>
<td>100250</td>
<td>5103</td>
<td>10</td>
</tr>
<tr>
<td>100250</td>
<td>6301</td>
<td>15</td>
</tr>
<tr>
<td>100250</td>
<td>6500</td>
<td>10</td>
</tr>
</tbody>
</table>

......

--- 71 row(s) selected.
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 92).
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 or a ROWS SINCE sequence function.

\[
\text{search-condition is:} \\
\quad \text{boolean-term} \; | \; \text{search-condition OR boolean-term}
\]

\[
\text{boolean-term is:} \\
\quad \text{boolean-factor} \; | \; \text{boolean-term AND boolean-factor}
\]

\[
\text{boolean-factor is:} \\
\quad [\text{NOT}] \; \text{boolean-primary}
\]

\[
\text{boolean-primary is:} \\
\quad \text{predicate} \; | \; (\text{search-condition})
\]

\text{OR}

specifies the resulting search condition is true if and only if either of the surrounding predicates or search conditions is true.

\text{AND}

specifies the resulting search condition is true if and only if both the surrounding predicates or search conditions are true.

\text{NOT}

reverses the truth value of its operand—the following predicate or search condition.

\text{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 151) 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 168).

Examples of Search Condition

- Select rows by using a search condition composed of three comparison predicates joined by AND operators:
• 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);
```

<table>
<thead>
<tr>
<th>PARTNUM</th>
<th>SUPPNUM</th>
<th>SUPPNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>212</td>
<td>1</td>
<td>NEW COMPUTERS INC</td>
</tr>
<tr>
<td>244</td>
<td>1</td>
<td>NEW COMPUTERS INC</td>
</tr>
<tr>
<td>255</td>
<td>1</td>
<td>NEW COMPUTERS INC</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7102</td>
<td>10</td>
<td>LEVERAGE INC</td>
</tr>
</tbody>
</table>

--- 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 74).

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 36). 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 19).

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 171).

Example of a Base Table

For example, this EMPLOYEE table is a base table in a sample database:

<table>
<thead>
<tr>
<th>EMPNUM</th>
<th>FIRST_NAME</th>
<th>LAST_NAME</th>
<th>DEPTNUM</th>
<th>JOBCODE</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ROGER</td>
<td>GREEN</td>
<td>9000</td>
<td>100</td>
<td>175500.00</td>
</tr>
<tr>
<td>23</td>
<td>JERRY</td>
<td>HOWARD</td>
<td>1000</td>
<td>100</td>
<td>137000.00</td>
</tr>
<tr>
<td>75</td>
<td>TIM</td>
<td>WALKER</td>
<td>3000</td>
<td>300</td>
<td>32000.00</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

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 47) and “DROP VIEW Statement” (page 55).

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:

<table>
<thead>
<tr>
<th>EMPNUM</th>
<th>FIRST_NAME</th>
<th>LAST_NAME</th>
<th>DEPTNUM</th>
<th>JOBCODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ROGER</td>
<td>GREEN</td>
<td>9000</td>
<td>100</td>
</tr>
<tr>
<td>23</td>
<td>JERRY</td>
<td>HOWARD</td>
<td>1000</td>
<td>100</td>
</tr>
<tr>
<td>75</td>
<td>TIM</td>
<td>WALKER</td>
<td>3000</td>
<td>300</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

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.
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 26).

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:

\[
\text{(FORMAT 'format-string') | (DATE, FORMAT 'format-string')}
\]

\textit{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:

\[
\text{(FORMAT 'format-string')}
\]

\textit{format-string} for other formats is:
- XXX

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 year/month/day.

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.
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.

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 74).

SAMPLE is a Trafodion SQL extension.

```sql
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:

```sql
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** \( \texttt{rows-size} \) \[\texttt{SORT BY colname [ASC[ENDING] | DESC[ENDING]] [colname [ASC[ENDING] | DESC[ENDING]]...]}\]

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 \( \texttt{rows-size} \), defined as:

\[\texttt{number-rows ROWS | BALANCE WHEN condition THEN number-rows ROWS [WHEN condition THEN number-rows ROWS]... [ELSE number-rows ROWS] END}\]

specifies the value of the size for FIRST sampling by using the number of rows intended in the sample. The value \( \texttt{number-rows} \) 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** \( \texttt{rows-size} \) \[\texttt{EVERY number-rows ROWS [SORT BY colname [ASC[ENDING] | DESC[ENDING]] [colname [ASC[ENDING] | DESC[ENDING]]...]}\]

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 \( \texttt{rows-size} \) 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:

\[\text{FLOOR} \left( \frac{N}{p} \right) \times s + \min \left( \text{MOD} \left( N, p \right), s \right)\]

\( \min \) 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:

```sql
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)
```
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:

  ```sql
  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.

  ```sql
  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.

  ```sql
  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
```
In this example, 62 rows exist in the SALESPEOPLE 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.

\[
\text{SELECT AVG(salary) FROM salesperson SAMPLE RANDOM 10 PERCENT;}
\]

(\text{EXPR})
----------------------
\text{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.

\[
\text{SELECT AVG(salary) FROM salesperson WHERE age > 35 SAMPLE RANDOM 10 PERCENT;}
\]

(\text{EXPR})
----------------------
\text{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 SALESPEOPLE and DEPARTMENT tables. You will get a different result each time you run this query because it is based on a random sample.

\[
\text{SELECT AVG(salary) FROM salesperson S, department D WHERE S.DNUM = D.DNUM AND D.NAME = 'CORPORATE' SAMPLE RANDOM 10 PERCENT;}
\]

(\text{EXPR})
----------------------
\text{106250.000}
--- 1 row(s) selected.

- In this example, the SALESPEOPLE 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.

\[
\text{SELECT AVG(salary) FROM ( SELECT salary, dnum FROM salesperson SAMPLE RANDOM 10 PERCENT ) AS S, department D WHERE S.DNUM = D.DNUM AND D.NAME = 'CORPORATE';}
\]
The results of this query and some of the results of previous queries might return null:

```sql
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';
```

--- 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 SALESPEOPLE 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.

```sql
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;
```

--- 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.

```sql
SELECT * 
FROM sales 
WHERE amount > 1000 
SAMPLE RANDOM 150 PERCENT;
```
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;
```

<table>
<thead>
<tr>
<th>AGE</th>
<th>SEX</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>male</td>
<td>28000.00</td>
</tr>
<tr>
<td>22</td>
<td>male</td>
<td>90000.00</td>
</tr>
<tr>
<td>22</td>
<td>female</td>
<td>136000.00</td>
</tr>
<tr>
<td>22</td>
<td>male</td>
<td>37000.40</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

--- 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;
```

<table>
<thead>
<tr>
<th>EMPID</th>
<th>PRODUCT</th>
<th>REGION</th>
<th>AMOUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>PCGOLD, 30MB</td>
<td>E</td>
<td>30000.00</td>
</tr>
<tr>
<td>23</td>
<td>PCDIAMOND, 60MB</td>
<td>W</td>
<td>40000.00</td>
</tr>
<tr>
<td>29</td>
<td>GRAPHICPRINTER, M1</td>
<td>N</td>
<td>11000.00</td>
</tr>
<tr>
<td>32</td>
<td>GRAPHICPRINTER, M2</td>
<td>S</td>
<td>15000.00</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>228</td>
<td>MONITORCOLOR, M2</td>
<td>N</td>
<td>10500.00</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

--- 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;
```

<table>
<thead>
<tr>
<th>EMPID</th>
<th>PRODUCT</th>
<th>REGION</th>
<th>AMOUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>PCGOLD, 30MB</td>
<td>E</td>
<td>30000.00</td>
</tr>
<tr>
<td>23</td>
<td>PCDIAMOND, 60MB</td>
<td>W</td>
<td>40000.00</td>
</tr>
<tr>
<td>29</td>
<td>GRAPHICPRINTER, M1</td>
<td>N</td>
<td>11000.00</td>
</tr>
<tr>
<td>32</td>
<td>GRAPHICPRINTER, M2</td>
<td>S</td>
<td>15000.00</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

--- 32 row(s) selected.
<table>
<thead>
<tr>
<th></th>
<th>DESCRIPTION</th>
<th>CODE</th>
<th>TYPE</th>
<th>PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>39</td>
<td>GRAPHICPRINTER, M3</td>
<td>S</td>
<td></td>
<td>20000.00</td>
</tr>
<tr>
<td>75</td>
<td>LASERPRINTER, X1</td>
<td>W</td>
<td></td>
<td>42000.00</td>
</tr>
</tbody>
</table>

--- 30 row(s) selected.
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 74).

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 198).

**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:

```sql
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 the grouping column, you must use a derived table for the SEQUENCE BY query and use the derived column in the GROUP BY clause:

```sql
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:

```sql
SELECT MOVINGSUM(avg_price,2)
FROM
  (SELECT ordernum, AVG(unit_price)
   FROM odetail)
```
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:

```sql
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:

```sql
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:

```sql
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;
```

<table>
<thead>
<tr>
<th>RCOUNT</th>
<th>MCOUNT</th>
<th>Part/Num</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>212</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>212</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>244</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>244</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>244</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>67</td>
<td>1</td>
<td>7301</td>
</tr>
<tr>
<td>68</td>
<td>2</td>
<td>7301</td>
</tr>
<tr>
<td>69</td>
<td>3</td>
<td>7301</td>
</tr>
<tr>
<td>70</td>
<td>4</td>
<td>7301</td>
</tr>
</tbody>
</table>

--- 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.)

```sql
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;

<table>
<thead>
<tr>
<th>Order/Num</th>
<th>MCOUNT</th>
<th>Part/Num</th>
<th>Order/Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>10250</td>
<td>1</td>
<td>244</td>
<td>2008-01-23</td>
</tr>
<tr>
<td>14000.00</td>
<td>14000.00</td>
<td>14000.00</td>
<td>2008-01-23</td>
</tr>
<tr>
<td>10250</td>
<td>2</td>
<td>5103</td>
<td>2008-01-23</td>
</tr>
<tr>
<td>4000.00</td>
<td>18000.00</td>
<td>18000.00</td>
<td>2008-01-23</td>
</tr>
<tr>
<td>10250</td>
<td>3</td>
<td>6500</td>
<td>2008-01-23</td>
</tr>
<tr>
<td>950.00</td>
<td>18950.00</td>
<td>18950.00</td>
<td>2008-01-23</td>
</tr>
<tr>
<td>20300</td>
<td>1</td>
<td>244</td>
<td>2008-02-06</td>
</tr>
<tr>
<td>28000.00</td>
<td>28000.00</td>
<td>46950.00</td>
<td>2008-02-06</td>
</tr>
<tr>
<td>20300</td>
<td>2</td>
<td>2001</td>
<td>2008-02-06</td>
</tr>
<tr>
<td>10000.00</td>
<td>38000.00</td>
<td>56950.00</td>
<td>2008-02-06</td>
</tr>
<tr>
<td>20300</td>
<td>3</td>
<td>2002</td>
<td>2008-02-06</td>
</tr>
<tr>
<td>14000.00</td>
<td>52000.00</td>
<td>70950.00</td>
<td>2008-02-06</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>80060</td>
<td>18</td>
<td>7102</td>
<td>2008-10-09</td>
</tr>
<tr>
<td>1650.00</td>
<td>187360.00</td>
<td>1113295.00</td>
<td></td>
</tr>
<tr>
<td>80060</td>
<td>19</td>
<td>7301</td>
<td>2008-10-09</td>
</tr>
<tr>
<td>5100.00</td>
<td>192460.00</td>
<td>1118395.00</td>
<td></td>
</tr>
</tbody>
</table>

--- 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 ;

<table>
<thead>
<tr>
<th>(EXPR)</th>
<th>Order/Date</th>
<th>(EXPR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2008-01-23</td>
<td>?</td>
</tr>
<tr>
<td>2</td>
<td>2008-02-06</td>
<td>14</td>
</tr>
<tr>
<td>3</td>
<td>2008-02-17</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>2008-03-03</td>
<td>14</td>
</tr>
<tr>
<td>5</td>
<td>2008-03-19</td>
<td>16</td>
</tr>
<tr>
<td>6</td>
<td>2008-03-19</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>2008-03-27</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>2008-04-10</td>
<td>14</td>
</tr>
<tr>
<td>9</td>
<td>2008-04-20</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>2008-05-12</td>
<td>22</td>
</tr>
<tr>
<td>11</td>
<td>2008-06-01</td>
<td>20</td>
</tr>
<tr>
<td>12</td>
<td>2008-07-21</td>
<td>50</td>
</tr>
<tr>
<td>13</td>
<td>2008-10-09</td>
<td>80</td>
</tr>
</tbody>
</table>

--- 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 \( \text{TRANSPOSE } (A,X),(B,Y),(C,Z) \text{ AS } (V1,V2) \), the expressions \( A, B, \) and \( C \) have compatible data types, and the expressions \( X, Y, \) and \( Z \) have compatible data types.

\( \text{transpose-col-list} \)

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.

\( \text{colname} \)

is an SQL identifier that specifies a column name. It identifies the column consisting of the values in \( \text{expression-list} \).

For example, in the transpose set \( \text{TRANSPOSE } A,B,C \text{ AS } V \), the column \( V \) corresponds to the values of the expressions \( A,B, \) and \( C \).

\( (\text{colname-list}) \)

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 \( \text{TRANSPOSE } (A,X),(B,Y),(C,Z) \text{ AS } (V1,V2) \), 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 \( \text{key-colname} \)

optionally specifies which expression (the value in the transpose column list corresponds to) by its position in the item list. \( \text{key-colname} \) is an SQL identifier. The data type of the key column is exact numeric, and the value is NOT NULL.

Considerations for \texttt{TRANSPOSE}

Multiple \texttt{TRANSPOSE} Clauses and Sets

- Multiple \texttt{TRANSPOSE} clauses can be used in the same query. For example:
  ```sql
  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 \texttt{TRANSPOSE} clause can contain multiple transpose sets. For example:
  ```sql
  SELECT KEYCOL1, VALCOL1, VALCOL2 FROM MYTABLE
  TRANSPOSE A, B, C AS VALCOL1
  D, E, F AS VALCOL2
  KEY BY KEYCOL
  ```

Degree and Column Order of the \texttt{TRANSPOSE} Result

The degree of the \texttt{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 \texttt{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 \texttt{TRANSPOSE} Result

The data type of each of the value columns is the union compatible data type of the corresponding expressions in the \texttt{transpose-item-list}. You cannot have expressions with data types that are not compatible in a \texttt{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 119).

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 \( v_i \) 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:

<table>
<thead>
<tr>
<th>columns-of-source</th>
<th>V1</th>
<th>V2</th>
<th>V3</th>
</tr>
</thead>
<tbody>
<tr>
<td>source-row 1</td>
<td>value-of-A</td>
<td>value-of-X</td>
<td>NULL</td>
</tr>
<tr>
<td>source-row 2</td>
<td>value-of-B</td>
<td>value-of-Y</td>
<td>NULL</td>
</tr>
<tr>
<td>source-row 3</td>
<td>value-of-C</td>
<td>value-of-Z</td>
<td>NULL</td>
</tr>
<tr>
<td>source-row 4</td>
<td>NULL</td>
<td>NULL</td>
<td>value-of-L</td>
</tr>
<tr>
<td>source-row 5</td>
<td>NULL</td>
<td>NULL</td>
<td>value-of-M</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>100</td>
<td>d1</td>
<td>e1</td>
<td>f1</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td>200</td>
<td>d2</td>
<td>e2</td>
<td>f2</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>KEYCOL</th>
<th>VALCOL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>100</td>
<td>d1</td>
<td>e1</td>
<td>f1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>100</td>
<td>d1</td>
<td>e1</td>
<td>f1</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>100</td>
<td>d1</td>
<td>e1</td>
<td>f1</td>
<td>3</td>
<td>100</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td>200</td>
<td>d2</td>
<td>e2</td>
<td>f2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td>200</td>
<td>d2</td>
<td>e2</td>
<td>f2</td>
<td>2</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td>200</td>
<td>d2</td>
<td>e2</td>
<td>f2</td>
<td>3</td>
<td>200</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>KEYCOL</th>
<th>VALCOL</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11</td>
<td>1</td>
<td>10</td>
<td>100</td>
</tr>
<tr>
<td>2</td>
<td>103</td>
<td>1</td>
<td>10</td>
<td>100</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>1</td>
<td>10</td>
<td>100</td>
</tr>
<tr>
<td>1</td>
<td>22</td>
<td>2</td>
<td>20</td>
<td>200</td>
</tr>
<tr>
<td>2</td>
<td>203</td>
<td>2</td>
<td>20</td>
<td>200</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>2</td>
<td>20</td>
<td>200</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>KEYCOL</th>
<th>VALCOL</th>
<th>D</th>
<th>COUNT(*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>d1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>d1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>100</td>
<td>d1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>d2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td>d2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>200</td>
<td>d2</td>
<td>1</td>
</tr>
</tbody>
</table>

- 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.

```sql
SELECT COUNT(DISTINCT VALCOL) FROM mytable
TRANSPOSE A, B, C AS VALCOL
KEY BY KEYCOL
GROUP BY KEYCOL;
```

--- 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:

```sql
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:

<table>
<thead>
<tr>
<th>KEYCOL1</th>
<th>VALCOL1</th>
<th>KEYCOL2</th>
<th>VALCOL2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>d1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>e1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>3</td>
<td>f1</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>1</td>
<td>d1</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>2</td>
<td>e1</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>3</td>
<td>f1</td>
</tr>
<tr>
<td>3</td>
<td>100</td>
<td>1</td>
<td>d1</td>
</tr>
<tr>
<td>3</td>
<td>100</td>
<td>2</td>
<td>e1</td>
</tr>
<tr>
<td>3</td>
<td>100</td>
<td>3</td>
<td>f1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>d2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>e2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>f2</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td>1</td>
<td>d2</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td>2</td>
<td>e2</td>
</tr>
</tbody>
</table>
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:

```
<table>
<thead>
<tr>
<th>KEYCOL</th>
<th>VALCOL1</th>
<th>VALCOL2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>?</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>?</td>
</tr>
<tr>
<td>3</td>
<td>100</td>
<td>?</td>
</tr>
<tr>
<td>4</td>
<td>?</td>
<td>d1</td>
</tr>
<tr>
<td>5</td>
<td>?</td>
<td>e1</td>
</tr>
<tr>
<td>6</td>
<td>?</td>
<td>f1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>?</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td>?</td>
</tr>
<tr>
<td>3</td>
<td>200</td>
<td>?</td>
</tr>
<tr>
<td>4</td>
<td>?</td>
<td>d2</td>
</tr>
<tr>
<td>5</td>
<td>?</td>
<td>e2</td>
</tr>
<tr>
<td>6</td>
<td>?</td>
<td>f2</td>
</tr>
</tbody>
</table>
```

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:

```
<table>
<thead>
<tr>
<th>KEYCOL</th>
<th>VALCOL1</th>
<th>VALCOL2</th>
<th>(EXPR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>?</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>?</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>100</td>
<td>?</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>?</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td>?</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>200</td>
<td>?</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>?</td>
<td>d2</td>
<td>1</td>
</tr>
</tbody>
</table>
```

192  SQL Clauses
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:

```sql
SELECT * FROM mytable
TRANSPOSE (1, A, 'abc'), (2, B, 'xyz')
AS (VALCOL1, VALCOL2, VALCOL3);
```

The result table of the TRANSPOSE query is:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>VALCOL1</th>
<th>VALCOL2</th>
<th>VALCOL3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>100</td>
<td>d1</td>
<td>e1</td>
<td>f1</td>
<td>1</td>
<td>1</td>
<td>abc</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>100</td>
<td>d1</td>
<td>e1</td>
<td>f1</td>
<td>2</td>
<td>10</td>
<td>xyz</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td>200</td>
<td>d2</td>
<td>e2</td>
<td>f2</td>
<td>1</td>
<td>2</td>
<td>abc</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td>200</td>
<td>d2</td>
<td>e2</td>
<td>f2</td>
<td>2</td>
<td>20</td>
<td>xyz</td>
</tr>
</tbody>
</table>
5 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 129).

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.

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>“AVG Function” (page 208)</td>
<td>Computes the average of a group of numbers derived from the evaluation of the expression argument of the function.</td>
</tr>
<tr>
<td>“COUNT Function” (page 228)</td>
<td>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).</td>
</tr>
<tr>
<td>“MAX/MAXIMUM Function” (page 274)</td>
<td>Determines a maximum value from the group of values derived from the evaluation of the expression argument.</td>
</tr>
<tr>
<td>“MIN Function” (page 275)</td>
<td>Determines a minimum value from the group of values derived from the evaluation of the expression argument.</td>
</tr>
<tr>
<td>“STDDEV Function” (page 324)</td>
<td>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.</td>
</tr>
<tr>
<td>“SUM Function” (page 328)</td>
<td>Computes the sum of a group of numbers derived from the evaluation of the expression argument of the function.</td>
</tr>
<tr>
<td>“VARIANCE Function” (page 339)</td>
<td>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.</td>
</tr>
</tbody>
</table>

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 83).

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.

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>“ASCII Function”</td>
<td>Returns the ASCII code value of the first character of a character expression.</td>
</tr>
<tr>
<td>“CHAR Function”</td>
<td>Returns the specified code value in a character set.</td>
</tr>
<tr>
<td>“CHAR_LENGTH Function”</td>
<td>Returns the number of characters in a string. You can also use CHARACTER_LENGTH.</td>
</tr>
<tr>
<td>“CODE_VALUE Function”</td>
<td>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.</td>
</tr>
<tr>
<td>“CONCAT Function”</td>
<td>Returns the concatenation of two character value expressions as a string value. You can also use the concatenation operator (</td>
</tr>
<tr>
<td>“INSERT Function”</td>
<td>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.</td>
</tr>
<tr>
<td>“LCASE Function”</td>
<td>Downshifts alphanumeric characters. You can also use LOWER.</td>
</tr>
<tr>
<td>“LEFT Function”</td>
<td>Returns the leftmost specified number of characters from a character expression.</td>
</tr>
<tr>
<td>“LOCATE Function”</td>
<td>Returns the position of a specified substring within a character string. You can also use POSITION.</td>
</tr>
<tr>
<td>“LOWER Function”</td>
<td>Downshifts alphanumeric characters. You can also use LCASE.</td>
</tr>
<tr>
<td>“LPAD Function”</td>
<td>Replaces the leftmost specified number of characters in a character expression with a padding character.</td>
</tr>
<tr>
<td>“LTRIM Function”</td>
<td>Removes leading spaces from a character string.</td>
</tr>
<tr>
<td>“OCTET_LENGTH Function”</td>
<td>Returns the length of a character string in bytes.</td>
</tr>
<tr>
<td>“POSITION Function”</td>
<td>Returns the position of a specified substring within a character string. You can also use LOCATE.</td>
</tr>
<tr>
<td>“REPEAT Function”</td>
<td>Returns a character string composed of the evaluation of a character expression repeated a specified number of times.</td>
</tr>
<tr>
<td>“REPLACE Function”</td>
<td>Returns a character string where all occurrences of a specified character string in the original string are replaced with another character string.</td>
</tr>
<tr>
<td>“RIGHT Function”</td>
<td>Returns the rightmost specified number of characters from a character expression.</td>
</tr>
<tr>
<td>“RPAD Function”</td>
<td>Replaces the rightmost specified number of characters in a character expression with a padding character.</td>
</tr>
<tr>
<td>“RTRIM Function”</td>
<td>Removes trailing spaces from a character string.</td>
</tr>
<tr>
<td>“SPACE Function”</td>
<td>Returns a character string consisting of a specified number of spaces.</td>
</tr>
<tr>
<td>“SUBSTRING/SUBSTR Function”</td>
<td>Extracts a substring from a character string.</td>
</tr>
</tbody>
</table>
### Translating Characters

- **TRANSLATE Function** (page 334)
  - Translates a character string from a source character set to a target character set.

- **TRIM Function** (page 335)
  - Removes leading or trailing characters from a character string.

- **UCASE Function** (page 336)
  - Upshifts alphanumeric characters. You can also use UPSHIFT or UPPER.

- **UPPER Function** (page 337)
  - Upshifts alphanumeric characters. You can also use UPSHIFT or UCASE.

- **UPSHIFT Function** (page 338)
  - 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 203)
  - Adds the integer number of months specified by `intr_expr` to `datetime_expr` and normalizes the result.

- **CONVERTTIMESTAMP Function** (page 225)
  - Converts a Julian timestamp to a TIMESTAMP value.

- **CURRENT Function** (page 230)
  - Returns the current timestamp. You can also use the “CURRENT_TIMESTAMP Function”.

- **CURRENT_DATE Function** (page 231)
  - Returns the current date.

- **CURRENT_TIME Function** (page 232)
  - Returns the current time.

- **CURRENT_TIMESTAMP Function** (page 233)
  - Returns the current timestamp. You can also use the “CURRENT Function”.

- **DATE_ADD Function** (page 234)
  - Adds the interval specified by `interval_expression` to `datetime_expr`.

- **DATE_PART Function (of an Interval)** (page 239)
  - 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 240)
  - 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 235)
  - Subtracts the specified `interval_expression` from `datetime_expr`.

- **DATE_TRUNC Function** (page 241)
  - Returns the date with the time portion of the day truncated.

- **DATEADD Function** (page 236)
  - Adds the interval specified by `datepart` and `num_expr` to `datetime_expr`.

- **DATEDIFF Function** (page 237)
  - Returns the integer value for the number of `datepart` units of time between `startdate` and `enddate`.

- **DATEFORMAT Function** (page 238)
  - Formats a datetime value for display purposes.

- **DAY Function** (page 242)
  - 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 243)
  - Returns the name of the day of the week from a date or timestamp expression.

- **DAYOFMONTH Function** (page 244)
  - 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 245)
  - Returns an integer value in the range 1 through 7 that represents the corresponding day of the week.
### Mathematical Functions

Use these mathematical functions within an SQL numeric value expression:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>“ABS Function”</td>
<td>Returns the absolute value of a numeric value expression.</td>
</tr>
<tr>
<td>“ACOS Function”</td>
<td>Returns the arccosine of a numeric value expression as an angle expressed in radians.</td>
</tr>
<tr>
<td>“ASIN Function”</td>
<td>Returns the arcsine of a numeric value expression as an angle expressed in radians.</td>
</tr>
<tr>
<td>“ATAN Function”</td>
<td>Returns the arctangent of a numeric value expression as an angle expressed in radians.</td>
</tr>
<tr>
<td>“ATAN2 Function”</td>
<td>Returns the arctangent of the x and y coordinates, specified by two numeric value expressions, as an angle expressed in radians.</td>
</tr>
<tr>
<td>“CEILING Function”</td>
<td>Returns the smallest integer greater than or equal to a numeric value expression.</td>
</tr>
<tr>
<td>“COS Function”</td>
<td>Returns the cosine of a numeric value expression, where the expression is an angle expressed in radians.</td>
</tr>
<tr>
<td>“COSH Function”</td>
<td>Returns the hyperbolic cosine of a numeric value expression, where the expression is an angle expressed in radians.</td>
</tr>
<tr>
<td>“DEGREES Function”</td>
<td>Converts a numeric value expression expressed in radians to the number of degrees.</td>
</tr>
<tr>
<td>“EXP Function”</td>
<td>Returns the exponential value (to the base e) of a numeric value expression.</td>
</tr>
<tr>
<td>“FLOOR Function”</td>
<td>Returns the largest integer less than or equal to a numeric value expression.</td>
</tr>
</tbody>
</table>
Returns the natural logarithm of a numeric value expression.

“LOG Function” (page 269)

Returns the base 10 logarithm of a numeric value expression.

“LOG10 Function” (page 270)

Returns the remainder (modulus) of an integer value expression divided by an integer value expression.

“MOD Function” (page 277)

Returns the value of the operand unless it is zero, in which case it returns NULL.

“NULLIFZERO Function” (page 290)

Returns the constant value of pi as a floating-point value.

“PI Function” (page 294)

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 Function” (page 296)

Converts a numeric value expression expressed in degrees to the number of radians.

“RADIANS Function” (page 298)

Returns the value of numeric_expr round to num places to the right of the decimal point.

“ROUND Function” (page 305)

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.

“SIGN Function” (page 319)

Returns the sine of a numeric value expression, where the expression is an angle expressed in radians.

“SIN Function” (page 320)

Returns the hyperbolic sine of a numeric value expression, where the expression is an angle expressed in radians.

“SINH Function” (page 321)

Returns the square root of a numeric value expression.

“SQRT Function” (page 323)

Returns the tangent of a numeric value expression, where the expression is an angle expressed in radians.

“TAN Function” (page 329)

Returns the hyperbolic tangent of a numeric value expression, where the expression is an angle expressed in radians.

“TANH Function” (page 330)

Returns the value of the operand unless it is NULL, in which case it returns zero.

“ZEROIFNULL Function” (page 343)

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 251) Calculates differences between values of a column expression in the current row and previous rows.

“DIFF2 Function” (page 253) Calculates differences between values of the result of DIFF1 of the current row and DIFF1 of previous rows.

Moving sequence functions:

“MOVINGAVG Function” (page 280) Returns the average of nonnull values of a column expression in the current window.
<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVINGCOUNT Function</td>
<td>Returns the number of nonnull values of a column expression in the current window.</td>
</tr>
<tr>
<td>MOVINGMAX Function</td>
<td>Returns the maximum of nonnull values of a column expression in the current window.</td>
</tr>
<tr>
<td>MOVINGMIN Function</td>
<td>Returns the minimum of nonnull values of a column expression in the current window.</td>
</tr>
<tr>
<td>MOVINGSTDEV Function</td>
<td>Returns the standard deviation of nonnull values of a column expression in the current window.</td>
</tr>
<tr>
<td>MOVINGSUM Function</td>
<td>Returns the sum of nonnull values of a column expression in the current window.</td>
</tr>
<tr>
<td>MOVINGVARIANCE Function</td>
<td>Returns the variance of nonnull values of a column expression in the current window.</td>
</tr>
</tbody>
</table>

**Offset sequence function:**

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OFFSET Function</td>
<td>Retrieves columns from previous rows.</td>
</tr>
</tbody>
</table>

**Running sequence functions:**

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RANK/RUNNINGRANK Function</td>
<td>Returns the rank of the given value of an intermediate result table ordered by a SEQUENCE BY clause in a SELECT statement.</td>
</tr>
<tr>
<td>RUNNINGAVG Function</td>
<td>Returns the average of nonnull values of a column expression up to and including the current row.</td>
</tr>
<tr>
<td>RUNNINGCOUNT Function</td>
<td>Returns the number of rows up to and including the current row.</td>
</tr>
<tr>
<td>RUNNINGMAX Function</td>
<td>Returns the maximum of values of a column expression up to and including the current row.</td>
</tr>
<tr>
<td>RUNNINGMIN Function</td>
<td>Returns the minimum of values of a column expression up to and including the current row.</td>
</tr>
<tr>
<td>RUNNINGRANK Function</td>
<td>Returns the rank of the given value of an intermediate result table ordered by a SEQUENCE BY clause in a SELECT statement.</td>
</tr>
<tr>
<td>RUNNINGSTDEV Function</td>
<td>Returns the standard deviation of nonnull values of a column expression up to and including the current row.</td>
</tr>
<tr>
<td>RUNNINGSUM Function</td>
<td>Returns the sum of nonnull values of a column expression up to and including the current row.</td>
</tr>
<tr>
<td>RUNNINGVARIANCE Function</td>
<td>Returns the variance of nonnull values of a column expression up to and including the current row.</td>
</tr>
</tbody>
</table>

**Other sequence functions:**

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LASTNOTNULL Function</td>
<td>Returns the last nonnull value for the specified column expression. If only null values have been returned, returns null.</td>
</tr>
<tr>
<td>ROWS SINCE Function</td>
<td>Returns the number of rows counted since the specified condition was last true.</td>
</tr>
<tr>
<td>ROWS SINCE CHANGED Function</td>
<td>Returns the number of rows counted since the specified set of values last changed.</td>
</tr>
<tr>
<td>THIS Function</td>
<td>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.</td>
</tr>
</tbody>
</table>

See “SEQUENCE BY Clause” (page 184) and the individual entry for each function.
Other Functions and Expressions

Use these other functions and expressions in an SQL value expression:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BITAND Function (page 210)</td>
<td>Performs ‘and’ operation on corresponding bits of the two operands.</td>
</tr>
<tr>
<td>CASE (Conditional) Expression (page 211)</td>
<td>A conditional expression. The two forms of the CASE expression are simple and searched.</td>
</tr>
<tr>
<td>CAST Expression (page 214)</td>
<td>Converts a value from one data type to another data type that you specify.</td>
</tr>
<tr>
<td>COALESCE Function (page 219)</td>
<td>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.</td>
</tr>
<tr>
<td>CONVERTTOHEX Function (page 223)</td>
<td>Converts the specified value expression to hexadecimal for display purposes.</td>
</tr>
<tr>
<td>DECODE Function (page 247)</td>
<td>Compares expr to each test_expr value one by one in the order provided.</td>
</tr>
<tr>
<td>EXPLAIN Function (page 256)</td>
<td>Generates a result table describing an access plan for a SELECT, INSERT, DELETE, or UPDATE statement.</td>
</tr>
<tr>
<td>ISNULL Function (page 263)</td>
<td>Returns the first argument if it is not null, otherwise it returns the second argument.</td>
</tr>
<tr>
<td>NULLIF Function (page 289)</td>
<td>Returns the value of the first operand if the two operands are not equal, otherwise it returns NULL.</td>
</tr>
<tr>
<td>NVL Function (page 291)</td>
<td>Returns the value of the first operand unless it is NULL, in which case it returns the value of the second operand.</td>
</tr>
</tbody>
</table>

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 136).

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 136).

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 130).

- `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 136).

- `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 194). See “Numeric Value Expressions” (page 136).

Examples of ADD_MONTHS

- This function returns the value DATE '2007-03-31':
  ```sql
  ADD_MONTHS (DATE '2007-02-28', 1, 1)
  ```

- This function returns the value DATE '2007-03-28':
  ```sql
  ADD_MONTHS (DATE '2007-02-28', 1, 0)
  ```

- This function returns the value DATE '2008-03-28':
  ```sql
  ADD_MONTHS (DATE '2008-02-28', 1, 1)
  ```

- This function returns the timestamp '2009-02-28 00:00:00':
  ```sql
  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.

```sql
ASCII (character-expression)
```

character-expression is an SQL character value expression that specifies a string of characters. See “Character Value Expressions” (page 129).

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:

```sql
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:

```sql
SELECT jobdesc, ASCII (jobdesc)
FROM persnl.job;
```

<table>
<thead>
<tr>
<th>JOBDESC</th>
<th>(EXPR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MANAGER</td>
<td>77</td>
</tr>
<tr>
<td>PRODUCTION SUPV</td>
<td>80</td>
</tr>
<tr>
<td>ASSEMBLER</td>
<td>65</td>
</tr>
<tr>
<td>SALESREP</td>
<td>83</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

--- 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 136).

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 136).

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.

\[
\text{ATAN2} \ (\text{numeric-expression-x}, \text{numeric-expression-y})
\]

\text{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 136).

**Example of ATAN2**

This function returns the value 2.66344329881899520E+000, or approximately 2.6634:

\[
\text{ATAN2} \ (1.192, -2.3)
\]
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 136) and “Interval Value Expressions” (page 133).

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:

```sql
SELECT AVG (salary)
FROM persnl.employee;
```

```
(EXPR)---------------------
              49441.52
```

--- 1 row(s) selected.

- Return the average value of the set of unique SALARY values:

```sql
SELECT AVG(DISTINCT salary) AS Avg_Distinct_Salary
FROM persnl.employee;
```

```
AVG_DISTINCT_SALARY---------------------
              53609.89
```

208 SQL Functions and Expressions
Return the average salary by department:

```
SELECT deptnum, AVG(salary) AS "AVERAGE SALARY"
FROM persnl.employee
WHERE deptnum < 3000
GROUP BY deptnum;
```

<table>
<thead>
<tr>
<th>Dept/Num</th>
<th>&quot;AVERAGE SALARY&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>52000.17</td>
</tr>
<tr>
<td>2000</td>
<td>50000.10</td>
</tr>
<tr>
<td>1500</td>
<td>41250.00</td>
</tr>
<tr>
<td>2500</td>
<td>37000.00</td>
</tr>
</tbody>
</table>

--- 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.

\[
\text{BITAND (expression, expression)}
\]

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 a 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

\[
\text{select bitand(1,3) from (values(1)) x(a);}\\
\]

(EXPR)

--------------

 1

--- 1 row(s) selected

\[
\text{select 1 & 3 from (values(1)) x(a);}\\
\]

(EXPR)

--------------

 1

--- 1 row(s) selected

\[
\text{select bitand(1,3) + 0 from (values(1)) x(a);}\\
\]

(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:

```sql
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:

```sql
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:

```sql
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'
END
```
WHEN 500 THEN 'ACCOUNTANT'
WHEN 600 THEN 'ADMINISTRATOR ANALYST'
WHEN 900 THEN 'SECRETARY'
ELSE NULL
END
FROM persnl.employee;

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>FIRST_NAME</th>
<th>(EXPR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GREEN</td>
<td>ROGER</td>
<td>MANAGER</td>
</tr>
<tr>
<td>HOWARD</td>
<td>JERRY</td>
<td>MANAGER</td>
</tr>
<tr>
<td>RAYMOND</td>
<td>JANE</td>
<td>MANAGER</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>CHOU</td>
<td>JOHN</td>
<td>SECRETARY</td>
</tr>
<tr>
<td>CONRAD</td>
<td>MANFRED</td>
<td>PROGRAMMER</td>
</tr>
<tr>
<td>HERMAN</td>
<td>JIM</td>
<td>SALESREP</td>
</tr>
<tr>
<td>CLARK</td>
<td>LARRY</td>
<td>ACCOUNTANT</td>
</tr>
<tr>
<td>HALL</td>
<td>KATHRYN</td>
<td>SYSTEM ANALYST</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

--- 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:

```sql
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;
```

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>FIRST_NAME</th>
<th>DEPTNUM</th>
<th>(EXPR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GREEN</td>
<td>ROGER</td>
<td>9000</td>
<td>193050.0000</td>
</tr>
<tr>
<td>HOWARD</td>
<td>JERRY</td>
<td>1000</td>
<td>153440.1120</td>
</tr>
<tr>
<td>RAYMOND</td>
<td>JANE</td>
<td>3000</td>
<td>136000.0000</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

--- 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.

\[
\text{CAST (} \{ \text{expression} \mid \text{NULL} \} \text{ AS data-type) }
\]

- \text{expression} | \text{NULL} specifies the operand to convert to the data type \text{data-type}.
  - If the operand is an \text{expression}, then \text{data-type} depends on the data type of \text{expression} and follows the rules outlined in “Valid Conversions for CAST” (page 214).
  - If the operand is \text{NULL}, or if the value of the \text{expression} is null, the result of \text{CAST} is \text{NULL}, regardless of the data type you specify.

- \text{data-type} specifies a data type to associate with the operand of \text{CAST}. See “Data Types” (page 117).

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 \text{CAST} of a numeric value to an INTERVAL type.
- Depending on how your file is set up, using \text{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:
  \[
  \text{CAST (123.956 as INTERVAL DAY(18))}
  \]

- This example returns the difference of two timestamps in minutes:
  \[
  \text{CAST((d.step_end - d.step_start) AS INTERVAL MINUTE)}
  \]

- 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.

```sql
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 136).

Example of CEILING

This function returns the integer value 3.0000000000000000E+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;

<table>
<thead>
<tr>
<th>CUSTNAME</th>
<th>(EXPR)</th>
<th>(EXPR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CENTRAL UNIVERSITY</td>
<td>67</td>
<td>C</td>
</tr>
<tr>
<td>BROWN MEDICAL CO</td>
<td>66</td>
<td>B</td>
</tr>
<tr>
<td>STEVENS SUPPLY</td>
<td>83</td>
<td>S</td>
</tr>
<tr>
<td>PREMIER INSURANCE</td>
<td>80</td>
<td>P</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

--- 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_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 129).

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.

\[
\text{COALESCE} (\text{expr1, expr2, ...})
\]

- \text{expr1}
  
  an expression to be compared.

- \text{expr2}
  
  an expression to be compared.

Example of COALESCE

COALESCE returns the value of the first operand that is not NULL:

\[
\text{SELECT COALESCE (office_phone, cell_phone, home_phone, pager, fax_num, '411') from emptbl;}
\]
**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:

```sql
>>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.

\[
\text{CONCAT (character-expr-1, character-expr-2)}
\]

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 129).

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 (||):

\[
\text{character-expr-1 || character-expr-2}
\]

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':

\[
\text{?p || ' 2002'}
\]

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:

\[
\text{CAST(?p AS CHAR(7)) || '2002'}
\]

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:

```sql
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)));

222 SQL Functions and Expressions
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:

  ```sql
  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;
  ```

<table>
<thead>
<tr>
<th>(EXPR)</th>
<th>(EXPR)</th>
<th>(EXPR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0025</td>
<td>7FFFFFFF</td>
<td>1FFFFFFFFFFFFFFF</td>
</tr>
</tbody>
</table>

- 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:

  ```sql
  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;
  ```

<table>
<thead>
<tr>
<th>(EXPR)</th>
<th>(EXPR)</th>
<th>(EXPR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>41424320</td>
<td>616263</td>
<td>0061006200630020</td>
</tr>
</tbody>
</table>

- Display the internal values for a DATE column, a TIME column, a TIMESTAMP(2) column, and a TIMESTAMP(6) column:

  ```sql
  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;
  ```

<table>
<thead>
<tr>
<th>(EXPR)</th>
<th>(EXPR)</th>
<th>(EXPR)</th>
<th>(EXPR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>07D8040F</td>
<td>0E201E</td>
<td>07D8040F0E201E0000000035</td>
<td>07D8040F0E201B00081ABB</td>
</tr>
</tbody>
</table>

- Display the internal values for an INTERVAL YEAR column, an INTERVAL YEAR(2) TO MONTH column, and an INTERVAL DAY TO SECOND column:

  ```sql
  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 '1' year, interval '3-2' year(2) to month, interval '1' year, interval '3-2' year(2) to month, 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;

<table>
<thead>
<tr>
<th>(EXPR)</th>
<th>(EXPR)</th>
<th>(EXPR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001</td>
<td>0026</td>
<td>0000027C2F9CB780</td>
</tr>
</tbody>
</table>
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 136).

**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.

\[ \text{COSH} \ (\text{numeric-expression}) \]

\text{numeric-expression}

is an SQL numeric value expression that specifies the value for the argument of the COSH function. See “Numeric Value Expressions” (page 136).

Example of COSH

This function returns the value 1.88842387716101568E+000, or approximately 1.8884, the hyperbolic cosine of 1.25 in radians:

\[ \text{COSH} \ (1.25) \]
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.

\[
\text{COUNT} \{(*) | ([\text{ALL} \mid \text{DISTINCT}] \text{ expression})\}
\]

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 129).

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)
---------
Count the number of distinct departments in the EMPLOYEE table:

```sql
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 233).

`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;

<table>
<thead>
<tr>
<th>Project/Description</th>
<th>(EXPR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SALT LAKE CITY</td>
<td>2008-01-18</td>
</tr>
<tr>
<td>ROSS PRODUCTS</td>
<td>2008-02-02</td>
</tr>
<tr>
<td>MONTANA TOOLS</td>
<td>2008-03-03</td>
</tr>
<tr>
<td>AHAUS TOOL/SUPPLY</td>
<td>2008-03-03</td>
</tr>
<tr>
<td>THE WORKS</td>
<td>2008-02-02</td>
</tr>
<tr>
<td>THE WORKS</td>
<td>2008-02-02</td>
</tr>
</tbody>
</table>

--- 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).

\[
\text{CURRENT\_TIME} \ [(\text{precision})]
\]

- \text{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:

\[
\text{INSERT INTO stats.logfile}
\text{(user_key, run_date, run_time, user_name)}
\text{VALUES (001, CURRENT\_DATE, CURRENT\_TIME, 'JuBroc');}
\]
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 230).

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;
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 194). 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.

\[
\text{DATE_ADD} (\text{datetime_expr}, \text{interval_expression})
\]

datetime_expr

is an expression that evaluates to a datetime value of type DATE or TIMESTAMP. See “Datetime Value Expressions” (page 130).

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 133).

Examples of DATE_ADD

- This function returns the value DATE '2007-03-07'
  \[
  \text{DATE_ADD(DATE '2007-02-28', INTERVAL '7' DAY)}
  \]

- This function returns the value DATE '2008-03-06'
  \[
  \text{DATE_ADD(DATE '2008-02-28', INTERVAL '7' DAY)}
  \]

- This function returns the timestamp '2008-03-07 00:00:00'
  \[
  \text{DATE_ADD(timestamp '2008-02-29 00:00:00', INTERVAL '7' DAY)}
  \]

- This function returns the timestamp '2008-02-28 23:59:59'
  \[
  \text{DATE_ADD(timestamp '2007-02-28 23:59:59', INTERVAL '12' MONTH)}
  \]

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 194). 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 130).

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 133).

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 194). 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:

<table>
<thead>
<tr>
<th>datepart</th>
<th>Abbreviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>YEAR</td>
<td>YY and YYYY</td>
</tr>
<tr>
<td>MONTH</td>
<td>M and MM</td>
</tr>
<tr>
<td>DAY</td>
<td>D and DD</td>
</tr>
<tr>
<td>HOUR</td>
<td>HH</td>
</tr>
<tr>
<td>MINUTE</td>
<td>MI and M</td>
</tr>
<tr>
<td>SECOND</td>
<td>SS and S</td>
</tr>
<tr>
<td>QUARTER</td>
<td>Q and QQ</td>
</tr>
<tr>
<td>WEEK</td>
<td>WW and WK</td>
</tr>
</tbody>
</table>

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 136).

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 130).

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

<table>
<thead>
<tr>
<th>datepart</th>
<th>Abbreviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>YEAR</td>
<td>YY and YYYY</td>
</tr>
<tr>
<td>MONTH</td>
<td>M and MM</td>
</tr>
<tr>
<td>DAY</td>
<td>D and DD</td>
</tr>
<tr>
<td>HOUR</td>
<td>HH</td>
</tr>
<tr>
<td>MINUTE</td>
<td>MI and M</td>
</tr>
<tr>
<td>SECOND</td>
<td>SS and S</td>
</tr>
<tr>
<td>QUARTER</td>
<td>Q and QQ</td>
</tr>
<tr>
<td>WEEK</td>
<td>WW and WK</td>
</tr>
</tbody>
</table>

startdate
    may be of type DATE or TIMESTAMP. See “Datetime Value Expressions” (page 130).

enddate
    may be of type DATE or TIMESTAMP. See “Datetime Value Expressions” (page 130).

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.
  

- 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 130).

DEFAULT | USA | EUROPEAN

specifies a format for a datetime value. See “Datetime Literals” (page 144).

**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:
  
  \[
  \text{DATEFORMAT (TIMESTAMP '2008-06-20 14:20:20.00', USA)}
  \]

  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:
  
  \[
  \text{DATEFORMAT (TIMESTAMP '2008-06-20 14:20:20.00', EUROPEAN)}
  \]

  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 133).

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 yyyy:mm
  - YEARQUARTER: Extracts the year and quarter, as a 5-digit integer of the form yyyy:q, 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 yyyy:ww. 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 "Datet ime Value Expressions" (page 130).

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.

\[
\text{DATE_TRUNC(text, datetime_expr)}
\]

- **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 130).

Examples of DATE_TRUNC

- This function returns the value of TIMESTAMP '2006-12-31 00:00:00'.
  
  \[
  \text{DATE_TRUNC('DAY', TIMESTAMP '2006-12-31 11:59:59')} \]

- This function returns the value of TIMESTAMP '2006-01-01 00:00:00'.
  
  \[
  \text{DATE_TRUNC('YEAR', TIMESTAMP '2006-12-31 11:59:59')} \]

- This function returns the value of TIMESTAMP '2006-12-01 00:00:00'.
  
  \[
  \text{DATE_TRUNC('MONTH', DATE '2006-12-31')} \]

- **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 130).

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;
```

<table>
<thead>
<tr>
<th>Start/Date</th>
<th>Time/Shipped</th>
<th>(EXPR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2008-04-10</td>
<td>2008-04-21 08:15:00.000000</td>
<td>10</td>
</tr>
</tbody>
</table>
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 130).

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;
```

<table>
<thead>
<tr>
<th>Start/Date</th>
<th>Time/Shipped</th>
<th>(EXPR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2008-04-10</td>
<td>2008-04-21 08:15:00.000000</td>
<td>Thursday</td>
</tr>
</tbody>
</table>
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 130).

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;

<table>
<thead>
<tr>
<th>Start/Date</th>
<th>Time/Shipped</th>
<th>(EXPR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2008-04-10</td>
<td>2008-04-21 08:15:00.000000</td>
<td>10</td>
</tr>
</tbody>
</table>
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 130).

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.

\[ \text{DAYOFYEAR (datetime-expression)} \]

- datetime-expression is an expression that evaluates to a datetime value of type DATE or TIMESTAMP. See “Datetime Value Expressions” (page 130).

Example of DAYOFYEAR

Return an integer that represents the day of the year from the START_DATE column in the PROJECT table:

\[
\text{SELECT start_date, ship_timestamp, DAYOFYEAR(start_date) }\\ \text{FROM persnl.project}\\ \text{WHERE projcode = 1000;}\\
\]

<table>
<thead>
<tr>
<th>Start/Date</th>
<th>Time/Shiped</th>
<th>(EXPR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2008-04-10</td>
<td>2008-04-21 08:15:00.000000</td>
<td>101</td>
</tr>
</tbody>
</table>
DECODE Function

The DECODE function compares \texttt{expr} to each \texttt{test_expr} value one by one in the order provided. If \texttt{expr} is equal to a \texttt{test_expr}, then the corresponding \texttt{retval} is returned. If no match is found, \texttt{default} is returned. If no match is found and \texttt{default} is omitted, NULL is returned.

DECODE is a Trafodion SQL extension.

\texttt{DECODE (expr, test_expr1, retval1 [, test_expr2, retval2 ... ] [, default ] )}

\texttt{expr}

is an SQL expression.

\texttt{test_expr1, test_expr2,..}

are each an SQL expression of a type comparable to that of \texttt{expr}.

\texttt{retval}

is an SQL expression.

\texttt{default, retval2,..}

are each an SQL expression of a type comparable to that of \texttt{retval}.

Considerations for DECODE

In a DECODE function, two nulls are considered to be equivalent. If \texttt{expr} is null, then the returned value is the \texttt{retval} of the first \texttt{test_expr} that is also null.

The \texttt{expr}, \texttt{test_expr}, \texttt{retval}, and \texttt{default} values can be derived from expressions.

The arguments can be any of the numeric types or character types. However, \texttt{expr} and each \texttt{test_expr} value must be of comparable types. If \texttt{expr} and \texttt{test_expr} values are character types, they must be in the same character set (to be comparable types.)

All the \texttt{retval} values and the \texttt{default} value, if any, must be of comparable types.

If \texttt{expr} and a \texttt{test_expr} value are character data, the comparison is made using nonpadded comparison semantics.

If \texttt{expr} and a \texttt{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 \texttt{INTEGER} and the other is \texttt{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 \texttt{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 (\texttt{REAL}, \texttt{FLOAT}, \texttt{DOUBLE PRECISION}, \texttt{NUMERIC}, or \texttt{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 \texttt{REAL}, \texttt{FLOAT}, or \texttt{DOUBLE PRECISION}, then the returned value is of type \texttt{DOUBLE PRECISION}.

If all the possible return values are of numeric types but not all the same, none are \texttt{REAL}, \texttt{FLOAT}, or \texttt{DOUBLE PRECISION}, and at least one is of type \texttt{NUMERIC}, then the returned value is of type \texttt{NUMERIC}.

If all the possible return values are of numeric types, none are \texttt{NUMERIC}, \texttt{REAL}, \texttt{FLOAT}, or \texttt{DOUBLE PRECISION}, and at least one is of type \texttt{DECIMAL}, then the returned value will be of type \texttt{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:

```sql
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:**

  ```sql
  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:**

  ```sql
  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 136).

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 184).

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;
```

<table>
<thead>
<tr>
<th>DIFF1_I1</th>
</tr>
</thead>
<tbody>
<tr>
<td>21959</td>
</tr>
<tr>
<td>-9116</td>
</tr>
<tr>
<td>-14461</td>
</tr>
<tr>
<td>7369</td>
</tr>
</tbody>
</table>
Retrieve the difference between the TS column in the current row and the TS column in the previous row:

```sql
SELECT DIFF1 (TS) AS DIFF1_TS
FROM mining.seqfcn
SEQUENCE BY TS;
```

<table>
<thead>
<tr>
<th>DIFF1_TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>3002620.000000</td>
</tr>
<tr>
<td>134157861.000000</td>
</tr>
<tr>
<td>168588029.000000</td>
</tr>
<tr>
<td>114055223.000000</td>
</tr>
</tbody>
</table>

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:

```sql
SELECT DIFF1 (I1,TS) AS DIFF1_I1TS
FROM mining.seqfcn
SEQUENCE BY TS;
```

<table>
<thead>
<tr>
<th>DIFF1_I1TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>.00007319</td>
</tr>
<tr>
<td>-.0000679</td>
</tr>
<tr>
<td>-.0000857</td>
</tr>
<tr>
<td>.0000646</td>
</tr>
</tbody>
</table>

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, 0.00007319 is equal to 21959 divided by 3002620.
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 184).

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 251).

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;
```

<table>
<thead>
<tr>
<th>DIFF2_I1</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
</tr>
<tr>
<td>?</td>
</tr>
<tr>
<td>-31075</td>
</tr>
<tr>
<td>-5345</td>
</tr>
<tr>
<td>21830</td>
</tr>
</tbody>
</table>

--- 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 251).

- Retrieve the difference in consecutive values of DIFF1(I1) divided by the difference in consecutive values of TS:

  ```sql
  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 136).

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:
  
  \[
  \text{EXP (1.25)}
  \]

- This function returns the value 2.0. The function EXP is the inverse of the function LOG:
  
  \[
  \text{EXP (LOG(2.0))}
  \]
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 257).

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 59).

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. To enable this feature, contact your HP Support representative for assistance.

**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.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MODULE_NAME</td>
<td>CHAR(60)</td>
<td>Reserved for future use.</td>
</tr>
<tr>
<td>STATEMENT_NAME</td>
<td>CHAR(60)</td>
<td>Statement name; truncated on the right if longer than 60 characters.</td>
</tr>
<tr>
<td>PLAN_ID</td>
<td>LARGEINT</td>
<td>Unique system-generated plan ID automatically assigned by Trafodion SQL; generated at compile time.</td>
</tr>
<tr>
<td>SEQ_NUM</td>
<td>INT</td>
<td>Sequence number of the current operator in the operator tree; indicates the sequence in which the operator tree is generated.</td>
</tr>
<tr>
<td>OPERATOR</td>
<td>CHAR(30)</td>
<td>Current operator type.</td>
</tr>
<tr>
<td>LEFT_CHILD_SEQ_NUM</td>
<td>INT</td>
<td>Sequence number for the first child operator of the current operator; null if node has no child operators.</td>
</tr>
<tr>
<td>RIGHT_CHILD_SEQ_NUM</td>
<td>INT</td>
<td>Sequence number for the second child operator of the current operator; null if node does not have a second child.</td>
</tr>
<tr>
<td>TNAME</td>
<td>CHAR(60)</td>
<td>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.</td>
</tr>
<tr>
<td>CARDINALITY</td>
<td>REAL</td>
<td>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.</td>
</tr>
<tr>
<td>OPERATOR_COST</td>
<td>REAL</td>
<td>Estimated cost associated with the current operator to execute the operator.</td>
</tr>
<tr>
<td>TOTAL_COST</td>
<td>REAL</td>
<td>Estimated cost associated with the current operator to execute the operator, including the cost of all subtrees in the operator tree.</td>
</tr>
<tr>
<td>DETAIL_COST</td>
<td>VARCHAR (200)</td>
<td>Cost vector of five items, described in the next table.</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>VARCHAR (3000)</td>
<td>Additional information about the operator.</td>
</tr>
</tbody>
</table>
The DETAIL_COST column of the EXPLAIN function results contains these cost factors:

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU_TIME</td>
<td>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.</td>
</tr>
<tr>
<td>IO_TIME</td>
<td>An estimate of the number of seconds of I/O time (seeks plus data transfer) to perform the I/O for this operator.</td>
</tr>
<tr>
<td>MSG_TIME</td>
<td>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.</td>
</tr>
<tr>
<td>IDLETIME</td>
<td>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.</td>
</tr>
<tr>
<td>PROBES</td>
<td>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.</td>
</tr>
</tbody>
</table>

Examples of EXPLAIN Function

Display the specified columns in the result table of the EXPLAIN function for the prepared statement REGION:

```sql
>>select seq_num, operator, operator_cost from table (explain (null, 'REG'));
```

<table>
<thead>
<tr>
<th>SEQ_NUM</th>
<th>OPERATOR</th>
<th>OPERATOR_COST</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>TRAFODION_SCAN</td>
<td>0.43691027</td>
</tr>
<tr>
<td>2</td>
<td>ROOT</td>
<td>0.0</td>
</tr>
</tbody>
</table>

--- 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.

\[ \text{EXTRACT (datetime-field FROM extract-source)} \]

\begin{itemize}
  \item \textit{datetime-field is:}
    \begin{itemize}
      \item YEAR
      \item MONTH
      \item DAY
      \item HOUR
      \item MINUTE
      \item SECOND
    \end{itemize}
  \item \textit{extract-source is:}
    \begin{itemize}
      \item datetime-expression
      \item interval-expression
    \end{itemize}

See “Datetime Value Expressions” (page 130) and “Interval Value Expressions” (page 133).

Examples of EXTRACT

- Extract the year from a DATE value:
  \[ \text{EXTRACT (YEAR FROM DATE '2007-09-28') \]}
  The result is 2007.
- Extract the year from an INTERVAL value:
  \[ \text{EXTRACT (YEAR FROM INTERVAL '01-09' YEAR TO MONTH) \]}
  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 136).

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 130).

**Example of HOUR**

Return an integer that represents the hour of the day from the SHIP_TIMESTAMP column in the PROJECT table:

```sql
SELECT start_date, ship_timestamp, HOUR(ship_timestamp)
FROM persnl.project
WHERE projcode = 1000;
```

<table>
<thead>
<tr>
<th>Start/Date</th>
<th>Time/Shipped</th>
<th>(EXPR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2007-04-10</td>
<td>2007-04-21</td>
<td>8</td>
</tr>
</tbody>
</table>
**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 129).

- **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:

```sql
UPDATE persnl.job
SET jobdesc = INSERT (jobdesc, 6, 3, ' REP')
WHERE jobdesc = 'SALESREP';
```

Now check the row you updated:

```sql
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.

\[
\text{JULIANTIMESTAMP (datetime-expression)}
\]

\[\text{datetime-expression}\]

is an expression that evaluates to a value of type DATE, TIME, or TIMESTAMP. If \[\text{datetime-expression}\] 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 130).

Considerations for JULIANTIMESTAMP

The \[\text{datetime-expression}\] 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:

  ```sql
  SELECT ship_timestamp, JULIANTIMESTAMP (ship_timestamp)
  FROM persnl.project
  WHERE projcode = 1000;
  ``

<table>
<thead>
<tr>
<th>SHIP_TIMESTAMP</th>
<th>(EXPR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2008-04-21 08:15:00.000000</td>
<td>212075525700000000</td>
</tr>
</tbody>
</table>

  --- 1 row(s) selected.

- Convert the DATE value into a Julian timestamp representation:

  ```sql
  SELECT start_date, JULIANTIMESTAMP (start_date)
  FROM persnl.project
  WHERE projcode = 1000;
  ``

<table>
<thead>
<tr>
<th>START_DATE</th>
<th>(EXPR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2008-04-10</td>
<td>212075454560000000</td>
</tr>
</tbody>
</table>

  --- 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 184).

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 271).

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 129).

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;
```

<table>
<thead>
<tr>
<th>(EXPR)</th>
<th>(EXPR)</th>
<th>(EXPR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Hotel Oregon</td>
<td>HOTEL OREGON</td>
<td>hotel oregon</td>
</tr>
</tbody>
</table>

--- 17 row(s) selected.

See “UCASE Function” (page 336).
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.

\[
\text{LEFT (character-expr, count)}
\]

- **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 129).

- **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':

  \[
  \text{LEFT ('Robert John Smith', 6)}
  \]

- 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 295).

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 129).

source-expression is an SQL character value expression that specifies the source string. The source-expression cannot be NULL. See “Character Value Expressions” (page 129).

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.

\[ \text{LOG (numeric-expression)} \]

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 136).

Example of LOG

This function returns the value 6.9314718059945344E-001, or approximately 0.69315:

\[ \text{LOG (2.0)} \]
**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 136).

**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 266).

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 129).

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;
```

```
<table>
<thead>
<tr>
<th>(EXPR)</th>
<th>(EXPR)</th>
<th>(EXPR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Hotel Oregon</td>
<td>HOTEL OREGON</td>
<td>hotel oregon</td>
</tr>
</tbody>
</table>
```

--- 17 row(s) selected.

See “UPPER Function” (page 337).
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.

\[
LPAD \left( str, \ len \ [, \ padstr] \right)
\]

- \(str\) can be an expression. See “Character Value Expressions” (page 129).
- \(len\) 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 335).

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.

**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 335) and “RTRIM Function” (page 310).
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.

\[
\text{MAX | MAXIMUM ([ALL | DISTINCT] expression)}
\]

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 129).

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:

\[
\text{SELECT MAX (salary) FROM persnl.employee;}
\]

(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 129).

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 130).

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;

<table>
<thead>
<tr>
<th>Start/Date</th>
<th>Time/Shipped</th>
<th>(EXPR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2008-04-10</td>
<td>2008-04-21 08:15:00.000000</td>
<td>15</td>
</tr>
</tbody>
</table>
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.

\[
\text{MOD (integer-expression-1, integer-expression-2)}
\]

**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 136).

Example of MOD

This function returns the value 2 as the remainder or modulus:

\[
\text{MOD (11, 3)}
\]
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 130).

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;

<table>
<thead>
<tr>
<th>Start/Date</th>
<th>Time/Shipped</th>
<th>(EXPR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2008-04-10</td>
<td>2008-04-21 08:15:00.000000</td>
<td>4</td>
</tr>
</tbody>
</table>
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 130).

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;
```

<table>
<thead>
<tr>
<th>Start/Date</th>
<th>Time/Shipped</th>
<th>(EXPR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2008-04-10</td>
<td>2008-04-21 08:15:00.000000</td>
<td>April</td>
</tr>
</tbody>
</table>
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 184).

MOVINGAVG is a Trafodion SQL extension.

```sql
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:

```sql
CREATE TABLE db.mining.seqfcn (I1 INTEGER, ts TIMESTAMP);
SELECT MOVINGAVG (I1, 3) AS MOVINGAVG3
FROM mining.seqfcn
SEQUENCE BY TS;
```

<table>
<thead>
<tr>
<th>I1</th>
<th>TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>6215</td>
<td>TIMESTAMP '1950-03-05 08:32:09'</td>
</tr>
<tr>
<td>28174</td>
<td>TIMESTAMP '1951-02-15 14:35:49'</td>
</tr>
<tr>
<td>null</td>
<td>TIMESTAMP '1955-05-18 08:40:10'</td>
</tr>
<tr>
<td>4597</td>
<td>TIMESTAMP '1960-09-19 14:40:39'</td>
</tr>
<tr>
<td>11966</td>
<td>TIMESTAMP '1964-05-01 16:41:02'</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MOVINGAVG3</th>
</tr>
</thead>
<tbody>
<tr>
<td>6215</td>
</tr>
<tr>
<td>17194</td>
</tr>
<tr>
<td>17194</td>
</tr>
<tr>
<td>16385</td>
</tr>
<tr>
<td>8281</td>
</tr>
</tbody>
</table>

--- 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 184).

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:

```sql
SELECT MOVINGCOUNT (I1, 3) AS MOVINGCOUNT3
FROM mining.seqfcn
SEQUENCE BY TS;
```

<table>
<thead>
<tr>
<th>MOVINGCOUNT3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

--- 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 184).

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 \((\text{integer-expression}) - 1\) 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 \(\text{integer-expression}\) and \(\text{max-rows}\).
- If these conditions are met, MOVINGMAX returns the same result as RUNNINGMAX:
  - The \(\text{integer-expression}\) is out of range, and \(\text{max-rows}\) is not specified. This condition includes the case in which both \(\text{integer-expression}\) and \(\text{max-rows}\) are larger than the result table.
  - The minimum of \(\text{integer-expression}\) and \(\text{max-rows}\) is out of range. In this case, \(\text{integer-expression}\) could be within range, but \(\text{max-rows}\) 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 184).

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 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 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 184).

MOVINGSTDDEV is a Trafodion SQL extension.

\[
\text{MOVINGSTDDEV(\text{column-expression}, \text{integer-expression}[, \text{max-rows}])}
\]

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 \((\text{integer-expression} - 1)\) 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 \(\text{integer-expression}\) and \(\text{max-rows}\).
- If these conditions are met, MOVINGSTDDEV returns the same result as RUNNINGSTDDEV:
  - The \(\text{integer-expression}\) is out of range, and \(\text{max-rows}\) is not specified. This condition includes the case in which both \(\text{integer-expression}\) and \(\text{max-rows}\) are larger than the result table.
  - The minimum of \(\text{integer-expression}\) and \(\text{max-rows}\) is out of range. In this case, \(\text{integer-expression}\) could be within range, but \(\text{max-rows}\) 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:

```sql
SELECT MOVINGSTDDEV (I1,3) AS MOVINGSTDDEV3
FROM mining.seqfcn
SEQUENCE BY TS;
```

<table>
<thead>
<tr>
<th>MOVINGSTDDEV3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0000000000000000E+000</td>
</tr>
<tr>
<td>1.5523578080753976E+004</td>
</tr>
<tr>
<td>1.4802016531456112E+004</td>
</tr>
<tr>
<td>1.5115012482076664E+004</td>
</tr>
<tr>
<td>6.03627542446499008E+003</td>
</tr>
</tbody>
</table>

--- 5 row(s) selected.

You can use the CAST function for display purposes. For example:

```sql
SELECT CAST(MOVINGSTDDEV (I1,3) AS DEC (18,3))
FROM mining.seqfcn
SEQUENCE BY TS;
```

<table>
<thead>
<tr>
<th>(EXPR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00000000000000000E+000</td>
</tr>
<tr>
<td>1.55235780807539760E+004</td>
</tr>
<tr>
<td>1.48020165314561120E+004</td>
</tr>
<tr>
<td>1.51150124820766640E+004</td>
</tr>
<tr>
<td>6.03627542446499008E+003</td>
</tr>
<tr>
<td>.000</td>
</tr>
<tr>
<td>15527.357</td>
</tr>
<tr>
<td>14802.016</td>
</tr>
<tr>
<td>15115.012</td>
</tr>
<tr>
<td>6036.275</td>
</tr>
</tbody>
</table>

--- 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 184).

MOVINGSUM is a Trafodion SQL extension.

\[
\text{MOVINGSUM} (\text{column-expression}, \text{integer-expression} [, \text{max-rows}])
\]

- **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 \((\text{integer-expression} - 1)\) 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 \(\text{integer-expression}\) and \(\text{max-rows}\).
- If these conditions are met, MOVINGSUM returns the same result as RUNNINGSUM:
  - The \(\text{integer-expression}\) is out of range, and \(\text{max-rows}\) is not specified. This condition includes the case in which both \(\text{integer-expression}\) and \(\text{max-rows}\) are larger than the result table.
  - The minimum of \(\text{integer-expression}\) and \(\text{max-rows}\) is out of range. In this case, \(\text{integer-expression}\) could be within range, but \(\text{max-rows}\) 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:

\[
\text{SELECT MOVINGSUM (I1,3) AS MOVINGSUM3}
\quad \text{FROM mining.seqfcn}
\quad \text{SEQUENCE BY TS;}
\]

<table>
<thead>
<tr>
<th>MOVINGSUM3</th>
</tr>
</thead>
<tbody>
<tr>
<td>6215</td>
</tr>
<tr>
<td>34389</td>
</tr>
<tr>
<td>34389</td>
</tr>
<tr>
<td>32771</td>
</tr>
<tr>
<td>16563</td>
</tr>
</tbody>
</table>

--- 5 row(s) selected.
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 184).

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)
--- 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.

\[
\text{NULLIF}(\text{expr1}, \text{expr2})
\]

- \text{expr1} an expression to be compared.
- \text{expr2} an expression to be compared.

The NULLIF(\text{expr1}, \text{expr2}) is equivalent to:

\[
\text{CASE } \text{WHEN } \text{expr1} = \text{expr2} \\
\text{THEN } \text{NULL} \\
\text{ELSE } \text{expr1} \\
\text{END}
\]

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 \text{value} is equal to 7. The return value is the value of the first argument when that value is not 7.

\[
\text{NULLIF}(\text{value}, 7)
\]
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.

\[
\text{NULLIFZERO (expression)}
\]

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.
  
  ```sql
  SELECT NULLIFZERO(salary) from employee_tab;
  ```

- This function returns a value of 1 for each row of the table:
  
  ```sql
  SELECT NULLIFZERO(1) from employee_tab;
  ```

- This function returns a value of NULL for each row of the table:
  
  ```sql
  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.

\[ \text{NVL} \left( \text{operand, new-operand} \right) \]

**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:
  ```sql
  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:
  ```sql
  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:
  ```sql
  select nvl(a1, 9999999) from t1;
  (EXPR)
  -----
  123
  34
  9999999
  --- 3 row(s) selected.
  ```
  ```sql
  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.

\[
\text{OCTET\_LENGTH} \ (\text{string-value-expression})
\]

- **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 129).

**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 292).

**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:

\[
\text{OCTET\_LENGTH} \ (\text{‘Robert’})
\]
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 184). 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.

```sql
PI()
```

**Example of PI**

This constant function returns the value 3.141592600000000000E+000:

```sql
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 268).

```
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 129).

*source-expression* is an SQL character value expression that specifies the source string. The *source-expression* cannot be NULL. See “Character Value Expressions” (page 129).

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)
```

are SQL numeric value expressions that specify the values for the base and exponent arguments of the POWER function. See “Numeric Value Expressions” (page 136).

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.

\[
\text{QUARTER (datetime-expression)}
\]

`datetime-expression` is an expression that evaluates to a datetime value of type DATE or TIMESTAMP. See “Datet ime Value Expressions” (page 130).

**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;
```

<table>
<thead>
<tr>
<th>Start/Date</th>
<th>Time/Shipped</th>
<th>(EXPR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2008-04-10</td>
<td>2008-04-21 08:15:00.000000</td>
<td>2</td>
</tr>
</tbody>
</table>
RADIANS Function

The RADIANS function converts a numeric value expression (expressed in degrees) to the number of radians.
RADIANS is a Trafodion SQL extension.

\[
\text{RADIANS (numeric-expression)} \\
\text{numeric-expression}
\]

is an SQL numeric value expression that specifies the value for the argument of the RADIANS function. See “Numeric Value Expressions” (page 136).

Examples of RADIANS

- Return the value 7.85398150000000000E-001, or approximately 0.78540 in degrees:
  \[
  \text{RADIANS (45)}
  \]

- Return the value 45 in degrees. The function DEGREES is the inverse of the function RADIANS.
  \[
  \text{DEGREES (RADIANS (45))}
  \]
**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.

\[
\text{RUNNINGRANK}(\text{expression}) \mid \text{RANK}(\text{expression})
\]

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:

\[
\text{RANK} = 1 \text{ for the first value of the intermediate result table.}
\]

\[
= \text{the previous value of \text{RANK} if the previous value of \text{expression} is the same as the current value of \text{expression}.}
\]

\[
= \text{RUNNINGCOUNT}(*) \text{ otherwise.}
\]

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:

<table>
<thead>
<tr>
<th>I1</th>
<th>I2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>3</td>
<td>200</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
</tr>
<tr>
<td>2</td>
<td>200</td>
</tr>
<tr>
<td>5</td>
<td>300</td>
</tr>
<tr>
<td>10</td>
<td>null</td>
</tr>
</tbody>
</table>
• Return the rank of $I_{1}$:

```sql
SELECT $I_{1}$, RUNNINGRANK ($I_{1}$) AS RANK
FROM cat.sch.seqfcn
SEQUENCE BY $I_{1}$;
```

<table>
<thead>
<tr>
<th>$I_{1}$</th>
<th>RANK</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>10</td>
<td>8</td>
</tr>
</tbody>
</table>

--- 8 row(s) selected.

• Return the rank of $I_{1}$ descending:

```sql
SELECT $I_{1}$, RUNNINGRANK ($I_{1}$) AS RANK
FROM cat.sch.seqfcn
SEQUENCE BY $I_{1}$ DESC;
```

<table>
<thead>
<tr>
<th>$I_{1}$</th>
<th>RANK</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
</tr>
</tbody>
</table>

--- 8 row(s) selected.

• Return the rank of $I_{2}$, using the alternative RANK syntax:

```sql
SELECT $I_{2}$, RANK ($I_{2}$) AS RANK
FROM cat.sch.seqfcn
SEQUENCE BY $I_{2}$;
```

<table>
<thead>
<tr>
<th>$I_{2}$</th>
<th>RANK</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>1</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
</tr>
<tr>
<td>200</td>
<td>3</td>
</tr>
<tr>
<td>200</td>
<td>3</td>
</tr>
<tr>
<td>200</td>
<td>3</td>
</tr>
<tr>
<td>300</td>
<td>6</td>
</tr>
<tr>
<td>?</td>
<td>7</td>
</tr>
<tr>
<td>?</td>
<td>7</td>
</tr>
</tbody>
</table>
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;
```

<table>
<thead>
<tr>
<th>I2</th>
<th>RANK</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td>1</td>
</tr>
<tr>
<td>?</td>
<td>1</td>
</tr>
<tr>
<td>300</td>
<td>3</td>
</tr>
<tr>
<td>200</td>
<td>4</td>
</tr>
<tr>
<td>200</td>
<td>4</td>
</tr>
<tr>
<td>200</td>
<td>4</td>
</tr>
<tr>
<td>100</td>
<td>7</td>
</tr>
<tr>
<td>100</td>
<td>7</td>
</tr>
</tbody>
</table>

--- 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;
```

<table>
<thead>
<tr>
<th>I2</th>
<th>RANK</th>
</tr>
</thead>
<tbody>
<tr>
<td>300</td>
<td>1</td>
</tr>
<tr>
<td>200</td>
<td>2</td>
</tr>
<tr>
<td>200</td>
<td>2</td>
</tr>
<tr>
<td>200</td>
<td>2</td>
</tr>
<tr>
<td>100</td>
<td>5</td>
</tr>
<tr>
<td>100</td>
<td>5</td>
</tr>
</tbody>
</table>

--- 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 129).

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 129).

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;
```

<table>
<thead>
<tr>
<th>Job Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>---------------</td>
</tr>
<tr>
<td>MANAGER COMNET</td>
</tr>
<tr>
<td>PRODUCTION COMNET</td>
</tr>
<tr>
<td>ASSEMBLER COMNET</td>
</tr>
<tr>
<td>SALESREP COMNET</td>
</tr>
<tr>
<td>SYSTEM ANAL COMNET</td>
</tr>
</tbody>
</table>

--- 10 row(s) selected.

```
UPDATE persnl.job 
SET jobdesc = REPLACE (jobdesc, 'COMNET', 'TDMNET');
```

<table>
<thead>
<tr>
<th>Job Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>---------------</td>
</tr>
<tr>
<td>MANAGER TDMNET</td>
</tr>
<tr>
<td>PRODUCTION TDMNET</td>
</tr>
<tr>
<td>ASSEMBLER TDMNET</td>
</tr>
<tr>
<td>SALESREP TDMNET</td>
</tr>
<tr>
<td>SYSTEM ANAL TDMNET</td>
</tr>
</tbody>
</table>

--- 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 129).

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':
  ```sql
  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:
  ```sql
  UPDATE persnl.job
  SET jobdesc = RIGHT (jobdesc, 12);
  ```
ROUND Function

The ROUND function returns the value of \texttt{numeric\_expr} rounded to \texttt{num} places to the right of the decimal point.

ROUND is a Trafodion SQL extension.

\[ \text{ROUND}(\text{numeric\_expr} \ [\ , \ \text{num} \ ]) \]

\textit{numeric\_expr} is an SQL numeric value expression.

\textit{num} specifies the number of places to the right of the decimal point for rounding. If \textit{num} is a negative number, all places to the right of the decimal point and \textit{num} places to the left of the decimal point are zeroed. If \textit{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 \textit{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 \textit{numeric\_expr} is rounded toward the nearest even number.

Examples of ROUND

- This function returns the value of 123.46.

  \[ \text{ROUND}(123.4567, 2) \]

- This function returns the value of 123.

  \[ \text{ROUND}(123.4567, 0) \]

- This function returns the value of 120.

  \[ \text{ROUND}(123.4567, -1) \]

- This function returns the value of 0.

  \[ \text{ROUND}(999.0, -4) \]

- This function returns the value of 1000.

  \[ \text{ROUND}(999.0, -3) \]

- This function returns the value of 2.0E+000.

  \[ \text{ROUND}(1.5E+000, 0) \]

- This function returns the value of 2.0E+00.

  \[ \text{ROUND}(2.5E+000, 0) \]

- This function returns the value of 1.0E+00.

  \[ \text{ROUND}(1.4E+000, 0) \]
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 184).

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.

c 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)
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

--- 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 184).

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 129).
- `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 335).

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 129).

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 335) and “LTRIM Function” (page 273).
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 184).

RUNNINGAVG is a Trafodion SQL extension.

\[
\text{RUNNINGAVG (column-expression)}
\]

- 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:

\[
\text{RUNNINGSUM(column-expr) / RUNNINGCOUNT(*)}
\]

Example of RUNNINGAVG

Return the average of nonnull values of I1 up to and including the current row:

\[
\begin{align*}
\text{SELECT RUNNINGAVG (I1) AS AVG_I1} \\
\text{FROM mining.seqfcn} \\
\text{SEQUENCE BY TS;}
\end{align*}
\]

<table>
<thead>
<tr>
<th>AVG_I1</th>
</tr>
</thead>
<tbody>
<tr>
<td>6215</td>
</tr>
<tr>
<td>17194</td>
</tr>
<tr>
<td>11463</td>
</tr>
<tr>
<td>9746</td>
</tr>
<tr>
<td>10190</td>
</tr>
</tbody>
</table>

--- 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 184).

RUNNINGCOUNT is a Trafodion SQL extension.

\[
\text{RUNNINGCOUNT \{\} | (column-expression)}
\]

\* as an argument causes RUNNINGCOUNT(*) to return the number of rows in the intermediate result table up to and including the current row.

\text{column-expression}

specifies a derived column determined by the evaluation of the column expression. If \text{column-expression} is the argument, RUNNINGCOUNT returns the number of rows containing nonnull values of \text{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 \text{I1} up to and including the current row:

\[
\begin{align*}
\text{SELECT} & \quad \text{RUNNINGCOUNT (I1) AS COUNT_I1} \\
& \quad \text{FROM mining.seqfcn} \\
& \quad \text{SEQUENCE BY TS;}
\end{align*}
\]

\begin{verbatim}
COUNT_I1
--------
 1
 2
 2
 3
 4
\end{verbatim}

--- 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 184).

RUNNINGMAX is a Trafodion SQL extension.

\[ \text{RUNNINGMAX (column-expression)} \]

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:

\[
\text{SELECT RUNNINGMAX (I1) AS MAX_I1}
\text{FROM mining.seqfcn}
\text{SEQUENCE BY TS;}
\]

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 184).

RUNNINGMIN is a Trafodion SQL extension.

\[ \text{RUNNINGMIN (column-expression)} \]

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:

\[
\text{SELECT RUNNINGMIN (I1) AS MIN_I1 }
\text{FROM mining.seqfcn }
\text{SEQUENCE BY TS;}
\]

\begin{verbatim}
MIN_I1
--------
 6215
 6215
 6215
 4597
 4597
\end{verbatim}

--- 5 row(s) selected.

RUNNINGRANK Function

See the “RANK/RUNNINGRANK Function” (page 299).
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 184).

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)
-----------------
.pt000
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 184).

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 184).

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
2.19099699999999968E+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 130).

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;

<table>
<thead>
<tr>
<th>Start/Date</th>
<th>Time/Shipped</th>
<th>(EXPR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2008-04-10</td>
<td>2008-04-21 08:15:00.000000</td>
<td>.000000</td>
</tr>
</tbody>
</table>
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 136).

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.

\[
\text{SIN} \left( \text{numeric-expression} \right)
\]

\text{numeric-expression}

is an SQL numeric value expression that specifies the value for the argument of the SIN function. See “Numeric Value Expressions” (page 136).

Example of SIN

This function returns the value 3.42052233254419840E-001, or approximately 0.3420, the sine of 0.3491 (which is 20 degrees):

\[
\text{SIN} \left( 0.3491 \right)
\]
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.

\[
\text{SINH} (\text{numeric-expression})
\]

numeric-expression

is an SQL numeric value expression that specifies the value for the argument of the SINH function. See “Numeric Value Expressions” (page 136).

Example of SINH

This function returns the value 1.60191908030082560E+000, or approximately 1.6019, the hyperbolic sine of 1.25:

\[
\text{SINH} (1.25)
\]
**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.

\[
\text{SPACE (} \text{length [}, \text{char-set-name}\text{)} }
\]

*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.

\[
\text{SQRT (numeric-expression)}
\]

\(\text{numeric-expression}\)

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 136).

Example of SQRT

This function returns the value 5.19615242270663232E+000, or approximately 5.196:

\[
\text{SQRT (27)}
\]
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 339).
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:

  ```sql
  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:

  ```sql
  SELECT STDDEV (price * qty_available) 
  FROM sales.parts;
  
  (EXPR)
  -------------------------
  7.1389949999999808E+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.

\[
\text{SUBSTRING (character-expr FROM start-position [FOR length])}
\]
or:

\[
\text{SUBSTRING (character-expr, start-position[,]length)}
\]

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 129).

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 \( \text{SUBSTRING (string FOR int) } \) equivalent to \( \text{SUBSTRING (string FROM 1 FOR int) } \). The Trafodion database software already supports the ANSI standard form as:
  \[ \text{SUBSTRING (string FROM int [ FOR int ]) } \]

- The SUBSTRING function treats \( \text{SUBSTRING (string, Fromint) } \) equivalent to \( \text{SUBSTRING (string FROM Fromint) } \). The Trafodion database software already supports \( \text{SUBSTRING (string, Fromint, Forint) } \) as equivalent to the ANSI standard form:
  \[ \text{SUBSTRING (string FROM Fromint FOR Forint) } \]

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.

\[
\text{SUM ([ALL | DISTINCT] expression)}
\]

**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 129).

**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:

- \( \text{SUM (SALARY)} \)
- \( \text{SUM (SALARY * 1.1)} \)
- \( \text{SUM (PARTCOST * QTY_ORDERED)} \)

**Example of SUM**

Compute the total value of parts in the current inventory:

\[
\text{SELECT SUM (price * qty_available) FROM sales.parts;}
\]

(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.

\[
\text{TAN} \ (\text{numeric-expression})
\]

\text{numeric-expression}

is an SQL numeric value expression that specifies the value for the argument of the TAN function. See “Numeric Value Expressions” (page 136).

Example of TAN

This function returns the value 3.64008908293626880E-001, or approximately 0.3640, the tangent of 0.3491 (which is 20 degrees):

\[
\text{TAN} \ (0.3491)
\]
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 136).

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 306).

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:

```sql
SELECT ROWS SINCE (THIS(I1) < I1) AS ROWS_SINCE_THIS
FROM mining.seqfcn
SEQUENCE BY TS;
```

<table>
<thead>
<tr>
<th>ROWS_SINCE_THIS</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
</tr>
<tr>
<td>?</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>?</td>
</tr>
</tbody>
</table>

--- 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 23:59:59.000000',
  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:

<table>
<thead>
<tr>
<th>Translation Name</th>
<th>Source Character Set</th>
<th>Target Character Set</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISO88591TOUTF8</td>
<td>ISO88591</td>
<td>UTF8</td>
<td>Translates ISO8859-1 characters to UTF8 characters. No data loss is possible.</td>
</tr>
<tr>
<td>UTF8TOISO88591</td>
<td>UTF8</td>
<td>ISO88591</td>
<td>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.</td>
</tr>
</tbody>
</table>

**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.

\[
\text{TRIM (}[\text{trim-type}] [\text{trim-char}] \text{ FROM} \text{ trim-source})
\]

- **trim-type** is: \( \text{LEADING} | \text{TRAILING} | \text{BOTH} \)
- **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 129).

**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':
  
  \[
  \text{TRIM ('} \text{ Robert }')
  \]

- 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:
  
  \[
  \text{TRIM (first_name)} || ' ' || \text{TRIM (last_name)}
  \]
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 337) or “UPSHIFT Function” (page 338).

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)

description-expression

is an SQL character value expression that specifies a string of characters to upshift. See “Character Value Expressions” (page 129).

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;
```

```
<table>
<thead>
<tr>
<th>EXPR</th>
<th>EXPR</th>
<th>EXPR</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Hotel Oregon</td>
<td>HOTEL OREGON</td>
<td>hotel oregon</td>
</tr>
</tbody>
</table>
```

--- 17 row(s) selected.

See “LCASE Function” (page 266).

For more examples of when to use the UCASE function, see “UPSHIFT Function” (page 338).
### 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 338) or “**UCASE Function**” (page 336).

**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 129).

### 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;
```

<table>
<thead>
<tr>
<th>(EXPR)</th>
<th>(EXPR)</th>
<th>(EXPR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Hotel Oregon</td>
<td>HOTEL OREGON</td>
<td>hotel oregon</td>
</tr>
</tbody>
</table>

--- 17 row(s) selected.

See “**LOWER Function**” (page 271).

For examples of when to use the **UPPER** function, see “**UPSHIFT Function**” (page 338).
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 337) or “UCASE Function” (page 336).

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 129).

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:

  ```sql
  SELECT UPSHIFT(custname), UPPER(custname), UCASE(custname)
  FROM sales.customer;
  ```

  ![Table](image)

  --- 17 row(s) selected.

- Perform a case-insensitive search for the DataSpeed customer:

  ```sql
  SELECT *
  FROM sales.customer
  WHERE UPSHIFT (custname) = 'DATASPEED';
  ```

  ![Table](image)

  --- 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:

  ```sql
  SELECT * FROM persnl.dept1 D1, persnl.dept2 D2
  WHERE UPSHIFT(D1.deptname) = UPSHIFT(D2.deptname);
  ```
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 $v_i$ 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, $w_i$ 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:

$$\text{VARIANCE} = \frac{\sum (v_i - \bar{v})^2}{N-1}$$

where $v_i$ is the $i$-th value of expression, $\bar{v}$ 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:

$$\text{VARIANCE} = \frac{\sum (v_i - \bar{v}_w)^2}{N}$$

where $v_i$ is the $i$-th value of expression, $w_i$ is the $i$-th value of weight, $\bar{v}_w$ 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 $\bar{v}_w$ of $v_i$ and $w_i$ is defined as:
where $v_i$ is the $i$-th value of expression, $w_i$ 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 $\text{SUM}(x^2)$ exceeds the value of $1.15792089237316192e77$ during the computation of $\text{VARIANCE}(x)$, then a numeric overflow occurs.

**Examples of VARIANCE**

- Compute the variance of the salary of the current employees:

  ```sql
  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:

  ```sql
  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.

\[
\text{WEEK (datetime-expression)}
\]

*datetime-expression* is an expression that evaluates to a datetime value of type DATE or TIMESTAMP. See “Datetime Value Expressions” (page 130).

**Example of WEEK**

Return an integer that represents the week of the year from the START_DATE column in the PROJECT table:

\[
\text{SELECT start_date, ship_timestamp, WEEK(start_date) FROM persnl.project WHERE projcode = 1000;}
\]

<table>
<thead>
<tr>
<th>Start/Date</th>
<th>Time/Shipped</th>
<th>(EXPR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2008-04-10</td>
<td>2008-04-21 08:15:00.000000</td>
<td>15</td>
</tr>
</tbody>
</table>
**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.

\[ \text{YEAR (datetime-expression)} \]

*datetime-expression* is an expression that evaluates to a datetime value of type DATE or TIMESTAMP. See “Datetime Value Expressions” (page 130).

**Example of YEAR**

Return an integer that represents the year from the START_DATE column in the PROJECT table:

```sql
SELECT start_date, ship_timestamp, YEAR(start_date)
FROM persnl.project
WHERE projcode = 1000;
```

<table>
<thead>
<tr>
<th>Start/Date</th>
<th>Time/Shipped</th>
<th>(EXPR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2008-04-10</td>
<td>2008-04-21 08:15:00.000000</td>
<td>2008</td>
</tr>
</tbody>
</table>
**ZEROIFNULL Function**

The ZEROIFNULL function returns a value of zero if the expression is NULL. Otherwise, it returns the value of the expression.

\[
\text{ZEROIFNULL (expression)}
\]

*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.

\[
\text{ZEROIFNULL (salary)}
\]
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 HP 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 3 Reserved SQL Identifiers — A**

<table>
<thead>
<tr>
<th>Action</th>
<th>Add</th>
<th>Admin</th>
<th>After</th>
<th>Aggregate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alias</td>
<td>All</td>
<td>Allocate</td>
<td>Alter</td>
<td>And</td>
</tr>
<tr>
<td>Any</td>
<td>Are</td>
<td>Array</td>
<td>As</td>
<td>Asc</td>
</tr>
<tr>
<td>Assertion</td>
<td>Async</td>
<td>At</td>
<td>Authorization</td>
<td>Avg</td>
</tr>
</tbody>
</table>

**Table 4 Reserved SQL Identifiers — B**

<table>
<thead>
<tr>
<th>Before</th>
<th>Begin</th>
<th>Between</th>
<th>Binary</th>
<th>Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit_length</td>
<td>Blob</td>
<td>Boolean</td>
<td>Both</td>
<td>Breadth</td>
</tr>
<tr>
<td>By</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 5 Reserved SQL Identifiers — C**

<table>
<thead>
<tr>
<th>Call</th>
<th>Cascade</th>
<th>Cascaded</th>
<th>Case</th>
<th>Cast</th>
</tr>
</thead>
<tbody>
<tr>
<td>Catalog</td>
<td>Char</td>
<td>Character</td>
<td>Character_len</td>
<td>Char_len</td>
</tr>
<tr>
<td>Check</td>
<td>Class</td>
<td>Clob</td>
<td>Close</td>
<td>Coalesce</td>
</tr>
<tr>
<td>Collate</td>
<td>Collation</td>
<td>Column</td>
<td>Commit</td>
<td>Completion</td>
</tr>
<tr>
<td>Connect</td>
<td>Connection</td>
<td>Constraint</td>
<td>Constraints</td>
<td>Constructor</td>
</tr>
<tr>
<td>Continue</td>
<td>Convert</td>
<td>Corresponding</td>
<td>Count</td>
<td>Create</td>
</tr>
<tr>
<td>Cross</td>
<td>Cube</td>
<td>Current</td>
<td>Current_date</td>
<td>Current_path</td>
</tr>
<tr>
<td>Current_role</td>
<td>Current_time</td>
<td>Current_timestamp</td>
<td>Current_user</td>
<td>Current_usr_intn</td>
</tr>
<tr>
<td>Cursor</td>
<td>Cycle</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 6 Reserved SQL Identifiers — D**

<table>
<thead>
<tr>
<th>Date</th>
<th>Datetime</th>
<th>Day</th>
<th>Deallocate</th>
<th>Dec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decimal</td>
<td>Declare</td>
<td>Default</td>
<td>Deferrable</td>
<td>Deferred</td>
</tr>
<tr>
<td>Delete</td>
<td>Depth</td>
<td>Deref</td>
<td>Desc</td>
<td>Describe</td>
</tr>
<tr>
<td>Descriptor</td>
<td>Destroy</td>
<td>Destructor</td>
<td>Deterministic</td>
<td>Diagnostics</td>
</tr>
<tr>
<td>Dictionary</td>
<td>Disconnect</td>
<td>Distinct</td>
<td>Domain</td>
<td>Double</td>
</tr>
<tr>
<td>Drop</td>
<td>Dynamic</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Table 7 Reserved SQL Identifiers — E</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------------------------------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EACH ELSE ELSEIF END END-EXEC</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EQUALS ESCAPE EXCEPT EXCEPTION EXEC</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXECUTE EXISTS EXTERNAL EXTRACT</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 8 Reserved SQL Identifiers — F</th>
</tr>
</thead>
<tbody>
<tr>
<td>FALSE FETCH FIRST FLOAT FOR</td>
</tr>
<tr>
<td>FOREIGN FOUND FRACTION FREE FROM</td>
</tr>
<tr>
<td>FULL FUNCTION</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 9 Reserved SQL Identifiers — G</th>
</tr>
</thead>
<tbody>
<tr>
<td>GENERAL GET GLOBAL GO GOTO</td>
</tr>
<tr>
<td>GRANT GROUP GROUPING</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 10 Reserved SQL Identifiers — H</th>
</tr>
</thead>
<tbody>
<tr>
<td>HAVING HOST HOUR</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 11 Reserved SQL Identifiers — I</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDENTITY IF IGNORE IMMEDIATE IN</td>
</tr>
<tr>
<td>INDICATOR INITIALLY INNER INOUT INPUT</td>
</tr>
<tr>
<td>INSENSITIVE INSERT INT INTEGER INTERSECT</td>
</tr>
<tr>
<td>INTERVAL INTO IS ISOLATION ITERATE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 12 Reserved SQL Identifiers — J</th>
</tr>
</thead>
<tbody>
<tr>
<td>JOIN</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 13 Reserved SQL Identifiers — K</th>
</tr>
</thead>
<tbody>
<tr>
<td>KEY</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 14 Reserved SQL Identifiers — L</th>
</tr>
</thead>
<tbody>
<tr>
<td>LANGUAGE LARGE LAST LATERAL LEADING</td>
</tr>
<tr>
<td>LEAVE LEFT LESS LEVEL LIKE</td>
</tr>
<tr>
<td>LIMIT LOCAL LOCALTIME LOCALTIMESTAMP LOCATOR</td>
</tr>
<tr>
<td>LOOP LOWER</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 15 Reserved SQL Identifiers — M</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAINTAIN MAP MATCH MATCHED MAX</td>
</tr>
<tr>
<td>MERGE MIN MINUTE MODIFIES MODIFY</td>
</tr>
<tr>
<td>MODULE MONTH</td>
</tr>
<tr>
<td>Table 16 Reserved SQL Identifiers — N</td>
</tr>
<tr>
<td>---------------------------------------</td>
</tr>
<tr>
<td>NAMES</td>
</tr>
<tr>
<td>NEW</td>
</tr>
<tr>
<td>NULL</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 17 Reserved SQL Identifiers — O</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCTET_LENGTH</td>
</tr>
<tr>
<td>ON</td>
</tr>
<tr>
<td>OPTIONS</td>
</tr>
<tr>
<td>OUT</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 18 Reserved SQL Identifiers — P</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAD</td>
</tr>
<tr>
<td>POSITION</td>
</tr>
<tr>
<td>PREPARE</td>
</tr>
<tr>
<td>PRIVILEGES</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 19 Reserved SQL Identifiers — Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>QUALIFY</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 20 Reserved SQL Identifiers — R</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ</td>
</tr>
<tr>
<td>REFERENCES</td>
</tr>
<tr>
<td>REPLACE</td>
</tr>
<tr>
<td>RETURNS</td>
</tr>
<tr>
<td>ROUTINE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 21 Reserved SQL Identifiers — S</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAVEPOINT</td>
</tr>
<tr>
<td>SECOND</td>
</tr>
<tr>
<td>SESSION_USER</td>
</tr>
<tr>
<td>SIMILAR</td>
</tr>
<tr>
<td>SPECIFIC_TYPE</td>
</tr>
<tr>
<td>SQL_DOUBLE</td>
</tr>
<tr>
<td>SQL_SMALLINT</td>
</tr>
<tr>
<td>SQLError</td>
</tr>
<tr>
<td>SUBSTRING</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 22 Reserved SQL Identifiers — T</th>
</tr>
</thead>
<tbody>
<tr>
<td>TABLE</td>
</tr>
<tr>
<td>THEN</td>
</tr>
</tbody>
</table>
### Table 22 Reserved SQL Identifiers — T (continued)

<table>
<thead>
<tr>
<th>Identifier</th>
<th>TO</th>
<th>TRAILING</th>
<th>TRANSACTION</th>
<th>TRANSLATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRANSLATION</td>
<td>TRANSPOSE</td>
<td>TREAT</td>
<td>TRIGGER</td>
<td>TRIM</td>
</tr>
<tr>
<td>TRUE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table 23 Reserved SQL Identifiers — U

<table>
<thead>
<tr>
<th>Identifier</th>
<th>UNION</th>
<th>UNIQUE</th>
<th>UNKNOWN</th>
<th>UNNEST</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNDER</td>
<td>UNION</td>
<td>UNIQUE</td>
<td>UNKNOWN</td>
<td>UNNEST</td>
</tr>
<tr>
<td>UPDATE</td>
<td>UPPER</td>
<td>UPHIFT</td>
<td>USAGE</td>
<td>USER</td>
</tr>
<tr>
<td>USING</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table 24 Reserved SQL Identifiers — V

<table>
<thead>
<tr>
<th>Identifier</th>
<th>VALUES</th>
<th>VARCHAR</th>
<th>VARIABLE</th>
<th>VARYING</th>
</tr>
</thead>
<tbody>
<tr>
<td>VALUE</td>
<td>VALUES</td>
<td>VARCHAR</td>
<td>VARIABLE</td>
<td>VARYING</td>
</tr>
<tr>
<td>VIEW</td>
<td>VIRTUAL</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table 25 Reserved SQL Identifiers — W

<table>
<thead>
<tr>
<th>Identifier</th>
<th>WHEN</th>
<th>WHenever</th>
<th>WHERE</th>
<th>WHILE</th>
</tr>
</thead>
<tbody>
<tr>
<td>WAIT</td>
<td>WHEN</td>
<td>WHenever</td>
<td>WHERE</td>
<td>WHILE</td>
</tr>
<tr>
<td>WITH</td>
<td>WITHOUT</td>
<td>WORK</td>
<td>WRITE</td>
<td></td>
</tr>
</tbody>
</table>

### Table 26 Reserved SQL Identifiers — Y

<table>
<thead>
<tr>
<th>Identifier</th>
<th>YEAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>YEAR</td>
<td></td>
</tr>
</tbody>
</table>

### Table 27 Reserved SQL Identifiers — Z

<table>
<thead>
<tr>
<th>Identifier</th>
<th>ZONE</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZONE</td>
<td></td>
</tr>
</tbody>
</table>
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 348), which defines the HBase interface.

HBASE_INTERFACE

<table>
<thead>
<tr>
<th>Category</th>
<th>HBase</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Interface to use to access HBase.</td>
</tr>
<tr>
<td>Values</td>
<td>Specify one of these values:</td>
</tr>
<tr>
<td></td>
<td>• JNI to use a JNI interface</td>
</tr>
<tr>
<td></td>
<td>• JNI_TRX to use a transactional interface with HBase-trx via JNI</td>
</tr>
<tr>
<td>The default value is JNI_TRX.</td>
<td></td>
</tr>
</tbody>
</table>

Managing Histograms

This section describes these CQDs that are used to manage histograms:

- “CACHE_HISTOGRAMS_REFRESH_INTERVAL” (page 348)
- “HIST_NO_STATS_REFRESH_INTERVAL” (page 349)
- “HIST_PREFETCH” (page 349)
- “HIST_ROWCOUNT_REQUIRING_STATS” (page 350)

CACHE_HISTOGRAMS_REFRESH_INTERVAL

<table>
<thead>
<tr>
<th>Category</th>
<th>Histograms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Defines the time interval after which timestamps for cached histograms are checked to be refreshed.</td>
</tr>
<tr>
<td>Values</td>
<td>Unsigned integer Unit is seconds. The default value is ‘3600’ (1 hour).</td>
</tr>
</tbody>
</table>

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.
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: System or 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_ROW_COUNT_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 350)
- “MDAM_SCAN_METHOD” (page 351)
- “SUBQUERY_UNNESTING” (page 351)

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 351)

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 352)
- "UPD_ORDERED" (page 353)

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
<table>
<thead>
<tr>
<th><strong>Category</strong></th>
<th>Influencing Query Plans</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
<td>Controls whether rows should be inserted, updated, or deleted in clustering key order.</td>
</tr>
<tr>
<td><strong>Values</strong></td>
<td></td>
</tr>
<tr>
<td>‘ON’</td>
<td>The optimizer generates and considers plans where the rows are inserted, updated, or deleted in clustering key order.</td>
</tr>
<tr>
<td>‘OFF’</td>
<td>The optimizer does not generate plans where the rows must be inserted, updated, or deleted in clustering key order.</td>
</tr>
<tr>
<td></td>
<td>The default value is ‘ON’.</td>
</tr>
</tbody>
</table>

**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 the 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
This appendix lists limits for various parts of Trafodion SQL.

<table>
<thead>
<tr>
<th>Category</th>
<th>Limit Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Column names</td>
<td>Up to 128 characters long, or 256 bytes of UTF8 text, whichever is less.</td>
</tr>
<tr>
<td>Schema names</td>
<td>Up to 128 characters long, or 256 bytes of UTF8 text, whichever is less.</td>
</tr>
<tr>
<td>Table names</td>
<td>ANSI names are of the form <code>schema.object</code>, where each part can be up to 128 characters long, or 256 bytes of UTF8 text, whichever is less.</td>
</tr>
</tbody>
</table>
Index

A
ABS function
  examples of, 201
  syntax diagram of, 201
Access options
  summary of, 20
  DELETE statement use of, 51
  DML statements use of, 19
  INSERT statement use of, 63
  READ COMMITTED, 20
  SELECT statement use of, 81
  UPDATE statement use of, 97
ACOS function
  examples of, 202
  syntax diagram of, 202
ADD_MONTHS function
  examples of, 203
  syntax diagram of, 203
Aggregate functions
  summary of, 194
  AVG, 208
  COUNT, 228
  DISTINCT clause, 83, 194
  MAX/MAXIMUM, 274
  MIN, 275
  STDDEV, 324
  SUM, 328
  VARIANCE, 339
Aliases, schemas, 165
ALTER TABLE statement
  authorization and availability requirements, 30
  considerations for, 30
  constraints implemented with indexes, 30
  DEFAULT clause, 173
  description for, 27
  examples of, 30
  syntax diagram of, 26
ANSI
  compliance, description of, 21
  names, schemas, 165
ASCII function
  examples of, 204
  syntax diagram of, 204
ASIN function
  examples of, 205
  syntax diagram of, 205
ATAN function
  examples of, 206
  syntax diagram of, 206
ATAN2 function
  examples of, 207
  syntax diagram of, 207
Authorization ID
  description of, 109
  AVG function
  DISTINCT clause use of, 208
  examples of, 208
  operand requirements, 208
  syntax diagram of, 208
B
BEGIN WORK statement
  considerations for, 31
  example of, 31
  syntax diagram of, 31
BETWEEN predicate
  examples of, 152
  logical equivalents, 152
  operand requirements, 151
  syntax diagram of, 151
BITAND function
  examples of, 210
  syntax diagram of, 210
BLOCK_TO_PREVENT_HALLOWEEN, 352
Boolean operators
  NOT, AND, OR, 166
  search condition use of, 166
C
CACHE_HISTOGRAMS_REFRESH_INTERVAL, 348
CASE expression
  data type of, 212
  examples of, 212
  searched CASE form, 211
  syntax diagram of, 211
CAST expression
  data type conversion, 214
  examples of, 215
  syntax diagram of, 214
  valid type combinations, 214
CEILING function
  examples of, 216
  syntax diagram of, 216
CHAR data type, 122, 123
CHAR function
  examples of, 217
  syntax diagram of, 217
CHAR VARYING data type, 122
CHAR_LENGTH function
  examples of, 218
  syntax diagram of, 218
Character sets
  setting default, 142
  supported in Trafodion SQL, 110
Character string data types
  CHAR and VARCHAR, differences, 123
Character string functions
  summary of, 195
  ASCII, 204
  CHAR, 217
  CHAR_LENGTH , 218
<table>
<thead>
<tr>
<th>Term</th>
<th>Page(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CODE_VALUE</td>
<td>220</td>
<td></td>
</tr>
<tr>
<td>CONCAT</td>
<td>221</td>
<td></td>
</tr>
<tr>
<td>INSERT</td>
<td>262</td>
<td></td>
</tr>
<tr>
<td>LCASE</td>
<td>266</td>
<td></td>
</tr>
<tr>
<td>LEFT</td>
<td>267</td>
<td></td>
</tr>
<tr>
<td>LOCATE</td>
<td>268</td>
<td></td>
</tr>
<tr>
<td>LOWER</td>
<td>271</td>
<td></td>
</tr>
<tr>
<td>LPAD</td>
<td>272</td>
<td></td>
</tr>
<tr>
<td>LTRIM</td>
<td>273</td>
<td></td>
</tr>
<tr>
<td>OCTET_LENGTH</td>
<td>292</td>
<td></td>
</tr>
<tr>
<td>POSITION</td>
<td>295</td>
<td></td>
</tr>
<tr>
<td>REPEAT</td>
<td>302</td>
<td></td>
</tr>
<tr>
<td>REPLACE</td>
<td>303</td>
<td></td>
</tr>
<tr>
<td>RIGHT</td>
<td>304</td>
<td></td>
</tr>
<tr>
<td>RPAD</td>
<td>309</td>
<td></td>
</tr>
<tr>
<td>RTRIM</td>
<td>310</td>
<td></td>
</tr>
<tr>
<td>SPACE</td>
<td>322</td>
<td></td>
</tr>
<tr>
<td>SUBSTRING/SUBSTR</td>
<td>326</td>
<td></td>
</tr>
<tr>
<td>TRANSLATE</td>
<td>334</td>
<td></td>
</tr>
<tr>
<td>TRIM</td>
<td>335</td>
<td></td>
</tr>
<tr>
<td>UCASE</td>
<td>336</td>
<td></td>
</tr>
<tr>
<td>UPPER</td>
<td>337</td>
<td></td>
</tr>
<tr>
<td>UPSHIFT</td>
<td>338</td>
<td></td>
</tr>
<tr>
<td>Character string literals</td>
<td>142</td>
<td></td>
</tr>
<tr>
<td>Character value expression</td>
<td>examples of, 130</td>
<td>syntax diagram of, 129</td>
</tr>
<tr>
<td>CHECK constraint</td>
<td>28, 29, 39, 113</td>
<td>description of, 113</td>
</tr>
<tr>
<td>Clauses</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEFAULT</td>
<td>173</td>
<td></td>
</tr>
<tr>
<td>FORMAT</td>
<td>175</td>
<td></td>
</tr>
<tr>
<td>SAMPLE</td>
<td>177</td>
<td></td>
</tr>
<tr>
<td>SEQUENCE BY</td>
<td>184</td>
<td></td>
</tr>
<tr>
<td>TRANPOSE</td>
<td>187</td>
<td></td>
</tr>
<tr>
<td>Clustering key</td>
<td>141</td>
<td></td>
</tr>
<tr>
<td>COALESCE function</td>
<td>219</td>
<td></td>
</tr>
<tr>
<td>syntax diagram of, 219</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CODE_VALUE function</td>
<td>220</td>
<td></td>
</tr>
<tr>
<td>syntax diagram of, 220</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Columns</td>
<td>111</td>
<td></td>
</tr>
<tr>
<td>column reference</td>
<td>111</td>
<td></td>
</tr>
<tr>
<td>default values</td>
<td>112</td>
<td></td>
</tr>
<tr>
<td>description of, 111</td>
<td></td>
<td></td>
</tr>
<tr>
<td>qualified name</td>
<td>111</td>
<td></td>
</tr>
<tr>
<td>COMMIT WORK statement</td>
<td></td>
<td></td>
</tr>
<tr>
<td>considerations for, 32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>example of, 32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>syntax diagram of, 32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Comparable data types</td>
<td>64, 119</td>
<td></td>
</tr>
<tr>
<td>Comparison predicates</td>
<td>153</td>
<td></td>
</tr>
<tr>
<td>comparison operators</td>
<td>153</td>
<td></td>
</tr>
<tr>
<td>data conversions</td>
<td>154</td>
<td></td>
</tr>
<tr>
<td>examples of, 154</td>
<td></td>
<td></td>
</tr>
<tr>
<td>operand requirements</td>
<td>153</td>
<td></td>
</tr>
<tr>
<td>syntax diagram of, 153</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compatible data types</td>
<td>64, 119</td>
<td></td>
</tr>
<tr>
<td>CONCAT function</td>
<td></td>
<td></td>
</tr>
<tr>
<td>examples of, 221</td>
<td></td>
<td></td>
</tr>
<tr>
<td>syntax diagram of, 221</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Concatenation operator (</td>
<td></td>
<td>)</td>
</tr>
<tr>
<td>examples of, 218, 221</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Concurrency</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DELETE statement</td>
<td>52</td>
<td></td>
</tr>
<tr>
<td>description of, 21</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INSERT statement</td>
<td>64</td>
<td></td>
</tr>
<tr>
<td>UPDATE statement</td>
<td>97</td>
<td></td>
</tr>
<tr>
<td>Constraints</td>
<td>28, 29, 39, 113</td>
<td>description of, 113</td>
</tr>
<tr>
<td>CHECK</td>
<td>28, 29, 39, 113</td>
<td>description of, 113</td>
</tr>
<tr>
<td>FOREIGN KEY</td>
<td>28, 29, 39, 113</td>
<td>description of, 113</td>
</tr>
<tr>
<td>NOT NULL</td>
<td>28, 38, 113</td>
<td>description of, 113</td>
</tr>
<tr>
<td>PRIMARY KEY</td>
<td>39, 113</td>
<td>description of, 113</td>
</tr>
<tr>
<td>UNIQUE</td>
<td>28, 29, 38, 113</td>
<td>description of, 113</td>
</tr>
<tr>
<td>CONTROL QUERY DEFAULT statement</td>
<td>considerations for, 33</td>
<td>examples of, 33</td>
</tr>
<tr>
<td>syntax diagram of, 33</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CONVERTTIMESTAMP function</td>
<td>examples of, 225</td>
<td>syntax diagram of, 225</td>
</tr>
<tr>
<td>CONVERTTOHEX function</td>
<td>examples of, 223</td>
<td>syntax diagram of, 223</td>
</tr>
<tr>
<td>Correlation names</td>
<td>114</td>
<td></td>
</tr>
<tr>
<td>examples of, 114</td>
<td></td>
<td></td>
</tr>
<tr>
<td>purpose of, 114</td>
<td></td>
<td></td>
</tr>
<tr>
<td>table reference use</td>
<td>114</td>
<td></td>
</tr>
<tr>
<td>COS function</td>
<td>226</td>
<td></td>
</tr>
<tr>
<td>syntax diagram of, 226</td>
<td></td>
<td></td>
</tr>
<tr>
<td>COSH function</td>
<td>227</td>
<td></td>
</tr>
<tr>
<td>syntax diagram of, 227</td>
<td></td>
<td></td>
</tr>
<tr>
<td>COUNT function</td>
<td>228</td>
<td></td>
</tr>
<tr>
<td>DISTINCT clause within, 228</td>
<td>examples of, 228</td>
<td>syntax diagram of, 228</td>
</tr>
<tr>
<td>CREATE INDEX statement</td>
<td>35</td>
<td></td>
</tr>
<tr>
<td>authorization requirments</td>
<td>35</td>
<td></td>
</tr>
<tr>
<td>considerations for, 34</td>
<td></td>
<td></td>
</tr>
<tr>
<td>examples of, 35</td>
<td></td>
<td></td>
</tr>
<tr>
<td>limits on indexes</td>
<td>35</td>
<td></td>
</tr>
<tr>
<td>syntax diagram of, 34</td>
<td></td>
<td></td>
</tr>
<tr>
<td>volatile index</td>
<td>34</td>
<td></td>
</tr>
<tr>
<td>CREATE TABLE AS statement</td>
<td>examples of, 45</td>
<td>syntax diagram of, 36</td>
</tr>
<tr>
<td>LOAD IF EXISTS</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>syntax diagram of, 36</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CREATE TABLE LIKE statement</td>
<td>43</td>
<td></td>
</tr>
<tr>
<td>COMPRESSSION attribute, 43</td>
<td>considerations for, 43</td>
<td>file attributes, 43</td>
</tr>
<tr>
<td>syntax diagram of, 36</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Datetime data types
- DATE, 123
description of, 123
examples of literals, 144
- TIME, 123
- TIMESTAMP, 123

Datetime functions
- summary of, 196
  - ADD_MONTHS, 203
  - CONVERT_TIMESTAMP, 225
  - CURRENT, 230
  - CURRENT_DATE, 231
  - CURRENT_TIME, 232
  - CURRENT_TIMESTAMP, 233
  - DATE_ADD, 234
  - DATE_PART (interval), 239
  - DATE_PART (timestamp), 240
  - DATE_SUB, 235
  - DATE_TRUNC, 236
  - DATEADD, 237
  - DATEDIFF, 238
  - DATEFORMAT, 239
  - DAY, 242
  - DAYNAME, 243
  - DAYOFMONTH, 244
  - DAYOFWEEK, 245
  - DAYOFYEAR, 246
  - EXTRACT, 247
  - HOUR, 261
  - JULIANTIMESTAMP, 264
  - MINUTE, 276
  - MONTH, 278
  - MONTHNAME, 279
  - QUARTER, 297
  - SECOND, 318
  - TIMESTAMPADD, 332
  - TIMESTAMPDIFF, 333
  - WEEK, 341
  - YEAR, 342

Datetime literals
- description of, 144
- syntax diagram of, 130
- examples of, 131

Datetime value expression
- examples of, 131
- syntax diagram of, 130
- DAY function
  - examples of, 242
  - syntax diagram of, 242
- DAYNAME function
  - examples of, 243
  - syntax diagram of, 243
- DAYOFMONTH function
  - examples of, 244
  - syntax diagram of, 244
- DAYOFWEEK function
  - examples of, 245
  - syntax diagram of, 245
- DAYOFYEAR function
  - examples of, 246
  - syntax diagram of, 246

DDL statements see Data Definition Language (DDL)
- statements
- DECIMAL data type, 127
- DECODE function
  - examples of, 248
  - syntax diagram of, 247
- DEFAULT clause
  - ALTER TABLE use of, 27
  - CREATE TABLE use of, 38, 173
  - examples of, 173
  - syntax diagram of, 173
- Default settings
  - changing, 33
- DEGREES function
  - examples of, 250
  - syntax diagram of, 250
- DELETE statement
  - access options, 51
  - authorization requirements, 51
  - considerations for, 51
  - examples of, 52
  - isolation levels, 52
  - syntax diagram of, 51
  - WHERE clause, 51
- Delimited identifiers, 139
- Derived column names
  - examples of, 111
  - syntax of, 111
- DETAIL_COST in EXPLAIN output
  - CPU_TIME, 258
  - IDLETIME, 258
  - IO_TIME, 258
  - MSG_TIME, 258
  - PROBES, 258
- DIFF1 function
  - equivalent definitions, 251
  - examples of, 251
  - syntax diagram of, 251
- DIFF2 function
  - equivalent definitions, 253
  - examples of, 253
  - syntax diagram of, 253
- DISTINCT clause
  - aggregate functions, 83, 194
  - AVG function use of, 208
  - COUNT function use of, 228
  - MAX function use of, 274
  - MAXIMUM function use of, 274
  - MIN function use of, 275
  - STDDEV function use of, 324
  - SUM function use of, 328
  - VARIANCE function use of, 339
- DML statements see Data Manipulation Language (DML)
  - statements
- DOUBLE PRECISION data type, 128
- DROP INDEX statement
  - authorization and availability requirements, 53
  - considerations for, 53
  - example of, 53
DROP TABLE statement  
authorization and availability requirements, 54  
considerations for, 54  
examples of, 54  
syntax diagram of, 54

DROP VIEW statement  
authorization and availability requirements, 55  
considerations for, 55  
example of, 55  
syntax diagram of, 55

DROP VOLATILE INDEX statement  
considerations for, 53  
extamples of, 53  
syntax diagram of, 53

DROP VOLATILE TABLE statement  
authorization requirements, 54  
extamples of, 54  
syntax diagram of, 54

Error messages, 23

EXECUTE statement  
considerations for, 57  
extamples of, 57  
scope of, 57  
syntax diagram of, 56

EXISTS predicate  
correlated subquery within, 156  
extamples of, 156  
syntax diagram of, 156

EXP function  
extamples of, 255  
syntax diagram of, 255

EXPLAIN function  
columns in result, 257  
extamples of, 258  
operator tree, 257  
plan in running query, 256  
syntax diagram of, 256

EXPLAIN statement  
displayed, 59  
extamples of, 60  
operators, 59  
OPTIONS 'f' considerations, 60  
plan in running query, 59  
reviewing query execution plans, 59  
syntax, 59

Expression  
character (or string) value, 129  
datetime value, 130, 131, 135  
description of, 129  
interval value, 131, 135  
numeric value, 136

Extended numeric precision, 120

FILE options  
CREATE TABLE use of, 36

Fixed-length character column, 123

FLOAT data type, 127

FLOOR function  
extamples of, 260  
syntax diagram of, 260

FOREIGN KEY constraint, 29, 39, 113

FORMAT clause  
considerations for date formats, 176  
considerations for other formats, 176  
extamples of, 176  
syntax diagram, 175

FULL join, description of, 79

Functions, ANSI compliant, 22

GET statement  
considerations for, 61  
extamples of, 61  
syntax diagram of, 61

HBASE_INTERFACE, 348

HIST_NO_STATS_REFRESH_INTERVAL, 349

HIST_PREFETCH, 349

HIST_ROWCOUNT_REQUIRING_STATS, 350

Histograms  
clearing, 100  
UPDATE STATISTICS use of, 101

HOUR function  
extamples of, 261  
syntax diagram of, 261

Identifiers, 139

IN predicate  
logical equivalent using ANY, 158  
operand requirements, 158  
syntax diagram of, 157

Index keys, 141

Indexes  
and transactions, 34  
CREATE INDEX statement, 34  
CREATE VOLATILE INDEX statement, 34  
description of, 140  
DROP INDEX statement, 53

INSERT function  
extamples of, 262  
syntax diagram of, 262

INSERT statement  
compatible data types, 64  
considerations for, 63  
extamples of, 65  
self-referencing, 63  
side tree, 35  
syntax diagram of, 63
target column list, 63
VALUES specification within, 64
INTEGER data type, 127
Interval data types
description of, 125
Interval literals
description of, 145
examples of, 147
Interval value expression
examples of, 134
syntax diagram of, 133
INVOKED statement
example of, 67
syntax diagram of, 67
ISNULL function
examples of, 263
syntax diagram of, 263
Isolation levels
READ COMMITTED, 21

I
Join
CROSS, 79
FULL, 79
JOIN ON, 79
join predicate, 87
LEFT, 79
limits, 83
NATURAL, 79
NATURAL FULL, 79
NATURAL LEFT, 79
NATURAL RIGHT, 79
optional specifications, 78
RIGHT, 79
types, 78
JOIN ON join, description of, 79
JOIN_ORDER_BY_USER, 350
JULIANTIMESTAMP function
examples of, 264
syntax diagram of, 264

K
Keys
clustering, 141
index, 141
primary, 141
SYSKEY, 141

L
LARGEINT data type, 127
LASTNOTNULL function
examples of, 265
syntax diagram of, 265
LCASE function
examples of, 266
syntax diagram of, 266
LEFT function
examples of, 267
syntax diagram of, 267
LEFT join, description of, 79
LIKE predicate
considerations for, 159
syntax of, 159
Limits
IN predicate, 157
indexes, 35
number of tables joined, 83
tables, 354
Literals
datetime, examples of, 144
description of, 142
examples of, 142
interval, examples of, 147
numeric, examples of, 148
Loading data into tables
LOAD IF EXISTS option, 44
LOCATE function
examples of, 268
result of, 268
syntax diagram of, 268
Locking
READ COMMITTED access option, 20
LOG function
examples of, 269
syntax diagram of, 269
LOG10 function
examples of, 270
syntax diagram of, 270
Logical name
Trafodion SQL objects, 116
Logical operators
NOT, AND, OR, 166
search condition use of, 166
LOWER function
examples of, 271
syntax diagram of, 271
LPAD function
examples of, 272
syntax diagram of, 272
LTRIM function
examples of, 273
syntax diagram of, 273

M
Magnitude, 138
Math functions
summary of, 197
ABS, 201
ACOS, 202
ASIN, 205
ATAN, 206
ATAN2, 207
CEILING, 216
COS, 226
COSH, 227
DEGREES, 250
EXP, 255
FLOOR, 260
LOG, 269
LOG10, 270
MOD, 277
NULLIFZERO, 290
PI, 294
POWER, 296
RADIANS, 298
ROUND, 305
SIGN, 319
SIN, 320
SINH, 321
SQRT, 323
TAN, 329
TANH, 330
ZEROIFNULL, 343

MAX function
considerations for, 274
DISTINCT clause within, 274
examples of, 274
syntax diagram of, 274

MAXIMUM function
considerations for, 274
DISTINCT clause within, 274
examples of, 274
syntax diagram of, 274

MDAM_SCAN_METHOD, 351

MERGE statement
considerations for, 68
description of, 68
example of, 70
merge from one table to another, 70
restrictions, 69
syntax diagram of, 68
upsert, single row, 68

MIN function
DISTINCT clause within, 275
examples of, 275
syntax diagram of, 275

MINUTE function
examples of, 276
syntax diagram of, 276

MOD function
examples of, 277
syntax diagram of, 277

MONTH function
examples of, 278
syntax diagram of, 278
MONTHNAME function
examples of, 279
syntax diagram of, 279

MOVINGAVG function
examples of, 280
syntax diagram of, 280

MOVINGCOUNT function
examples of, 281
syntax diagram of, 281

MOVINGMAX function
examples of, 282
syntax diagram of, 282

MOVINGMIN function
examples of, 283
syntax diagram of, 283

MOVINGSTDDEV function
examples of, 284
syntax diagram of, 284

MOVINGSUM function
examples of, 286
syntax diagram of, 286

MOVINGVARIANCE function
examples of, 287
syntax diagram of, 287

NATIONAL CHAR data type, 122
NATIONAL CHAR VARYING data type, 122

NATURAL FULL join, description of, 79
NATURAL join, description of, 79
NATURAL LEFT join, description of, 79
NATURAL RIGHT join, description of, 79
NCHAR data type, 122
NCHAR VARYING data type, 122

NULL predicate
examples of, 161
syntax diagram of, 161

NULL, using, 96

NULLIF function
examples of, 289
syntax diagram of, 289

NULLIFZERO function
examples of, 290
syntax diagram of, 290

Numeric data types
approximate numeric, 127
exact numeric, 127
extended numeric, 120
literals, examples of, 148

Numeric literals
approximate, 147
exact, 147
examples of, 148

Numeric value expression
evaluation order, 137
examples of, 138
syntax diagram of, 136

NVL function
examples of, 291
syntax diagram of, 291

Object names, 116
Object namespace, 116

Objects
description of, 116
logical names, 116

OCTET_LENGTH function
Q
Quantified comparison predicates
ALL, ANY, SOME, 162
examples of, 163
operand requirements, 162
result of, 163
syntax diagram of, 162
QUARTER function
examples of, 297
syntax diagram of, 297
Query execution plan
displayed, 59
in running query, 59
operators, 59
reviewing, 59
Query expression
SELECT statement use of, 77
Query specification
SELECT statement use of, 79
simple table, form of, 79
R
RADIANS function
examples of, 298
syntax diagram of, 298
RANK/RUNNINGRANK function
example for, 299
syntax diagram of, 299
READ COMMITTED, 20
REAL data type, 128
Referential integrity
FOREIGN KEY constraint, 28, 29, 39
REPEAT function
examples of, 302
syntax diagram of, 302
REPLACE function
examples of, 303
syntax diagram of, 303
Reserved
words, Trafodion SQL, 344
Resource control statements
EXECUTE statement, 56
PREPARE statement, 71
UPDATE STATISTICS statement, 100
RIGHT function
examples of, 304
syntax diagram of, 304
RIGHT join, description of, 79
ROLLBACK WORK statement
considerations for, 73
example of, 73
syntax diagram of, 73
ROUND function
examples of, 305
syntax diagram of, 305
Row value constructor
BETWEEN predicate use of, 151
comparison predicates use of, 153
IN predicate use of, 157
NULL predicate use of, 161
quantified comparison predicates use of, 162
ROWS SINCE CHANGED function
considerations for, 308
examples of, 308
syntax diagram of, 308
ROWS SINCE function
examples of, 306
syntax diagram of, 306
RPAD function
examples of, 309
syntax diagram of, 309
RTRIM function
examples of, 310
syntax diagram of, 310
RUNNINGAVG function
 equivalent definition, 311
examples of, 311
syntax diagram of, 311
RUNNINGCOUNT function
examples of, 312
syntax diagram of, 312
RUNNINGMAX function
examples of, 313
syntax diagram of, 313
RUNNINGMIN function
examples of, 314
syntax diagram of, 314
RUNNINGSTDDEV function
 equivalent definition, 315
examples of, 315
syntax diagram of, 315
RUNNINGSUM function
examples of, 316
syntax diagram of, 316
RUNNINGVARIANCE function
examples of, 317
syntax diagram of, 317
S
SAMPLE clause
examples of, 178
SELECT statement use of, 177
syntax diagram of, 177
Scale, 138
SCHEMA, 351
Schemas
description of, 165
Search condition
Boolean operators within, 166
CASE expression use of, 212
DELETE statement use of, 51
description of, 168
examples of, 166
predicate within, 166
syntax diagram of, 166
UPDATE statement use of, 97
SECOND function
examples of, 318
syntax diagram of, 318
SELECT statement
access options, 81
authorization requirements, 82
considerations for, 82
DISTINCT clause, 76
embedded delete, 77
embedded insert, 78
embedded update, 77, 78
examples of, 86
FROM clause, 76
GROUP BY clause, 81, 84
HAVING clause, 81
joined table within, 78
LIMIT clause, 82
limit on join tables, 83
ORDER BY clause, 82, 84
RETURN list, 77
select list elements, 76
SEQUENCE BY clause, 80
simple table within, 79
syntax diagram of, 74
TRANSPOSE clause, 80
UNION ALL operation, 85
union operation within, 82, 84
views and, 82
WHERE clause, 79
Self-referencing INSERT, 64
SEQUENCE BY clause
examples of, 185
SELECT statement use of, 184
syntax diagram of, 184
Sequence functions
summary of, 198
DIFF1, 251
DIFF2, 253
LASTNOTNULL, 265
MOVINGAVG, 280
MOVINGCOUNT, 281
MOVINGMAX, 282
MOVINGMIN, 283
MOVINGSTDDEV, 284
MOVINGSUM, 286
MOVINGVARIANCE, 287
OFFSET, 293
RANK/RUNNINGRANK, 299
ROWS SINCE, 306
ROWS SINCE CHANGED, 308
RUNNINGAVG, 311
RUNNINGCOUNT, 312
RUNNINGMAX, 313
RUNNINGMIN, 314
RUNNINGSTDDEV, 315
RUNNINGSUM, 316
RUNNINGVARIANCE, 317
THIS, 331
Set functions, 194
SET PARAM command, 56
SET SCHEMA statement
considerations for, 92
example of, 92
syntax diagram of, 92
SET TRANSACTION statement
  autocommit option, 93
  considerations for, 93
  examples of, 93
  explicit transactions, 93
  implicit transactions, 93
  syntax diagram of, 93
Side tree insert
  definition of, 35
SIGN function
  examples of, 319
  syntax diagram of, 319
Simple table, in SELECT statement, 79
SIN function
  examples of, 320
  syntax diagram of, 320
SINH function
  examples of, 321
  syntax diagram of, 321
SMALLINT data type, 127
SPACE function
  examples of, 322
  syntax diagram of, 322
SQL statements
  ANSI compliant, 22
  Trafodion SQL extensions, 22
SQL value expression, 129
SQRT function
  examples of, 323
  syntax diagram of, 323
Statements, SQL
  ANSI compliant, 22
  Trafodion SQL extensions, 22
Statistics
  clearing, 100
  UPDATE STATISTICS statement, 102, 103
STDDEV function
  DISTINCT clause within, 324
  examples of, 325
  statistical definition of, 324
  syntax diagram of, 324
STORE BY clause, 37
Stored text
  reserved words, 344
String literals, 142
String value expression
  examples of, 130
  syntax diagram of, 129
Subquery
  correlated, 156, 168
  description of, 168
  inner query, 168
  outer query, 168
  outer reference, 168
  row
    BETWEEN predicate, 151
  comparison predicate, 153
  IN predicate, 157
  NULL predicate, 161
  quantified comparison predicate, 162
  UPDATE statement, 97
  UPSERT SELECT statement, 105
scalar
  BETWEEN predicate, 151
  comparison predicate, 153
  IN predicate, 157
  NULL predicate, 161
  quantified comparison predicate, 162
table, 157
SUBQUERY_UNNESTING, 351
SUBSTR function
  examples of, 327
  operand requirements, 326
  syntax diagram of, 326
SUBSTRING function
  examples of, 327
  operand requirements, 326
  syntax diagram of, 326
Suitable keys
  guidelines for selecting, 41
SUM function
  DISTINCT clause within, 328
  examples of, 328
  syntax diagram of, 328
SYSKEY
  system-defined clustering key, 141
T
Table
  creating, 36
  description of, 170
  dropping, 54
  limits, 354
  subquery, 157
Table reference
  description of, 77
  SELECT statement use of, 76
TABLE statement
  considerations for, 95
  examples, 95
  syntax diagram for, 95
Table value constructor
  description of, 79
  simple table, form of, 79
TAN function
  examples of, 329
  syntax diagram of, 329
TANH function
  examples of, 330
  syntax diagram of, 330
THIS function
  examples of, 331
  syntax diagram of, 331
TIMESTAMPADD function
  description, 332
examples, 332
TIMESTAMPDIFF function
description, 333
examples, 333
Trafodion SQL objects, logical names, 116
Transaction control statements
BEGIN WORK, 31
COMMIT WORK, 32
ROLLBACK WORK, 73
SET TRANSACTION statement, 93
Transaction isolation levels
READ COMMITTED, 21
Transaction management, 20
AUTO COMMIT, effect of, 21
BEGIN WORK, 31
COMMIT WORK, 32
ROLLBACK WORK, 73
rules for DML statements, 21
SET TRANSACTION, 93
TRANSLATE function, syntax diagram of, 334
TRANSPOSE clause
cardinality of result, 189
degree of result, 188
examples of, 189
SELECT statement use of, 187
syntax diagram of, 187
TRIM function
examples of, 335
syntax diagram of, 335
U
UCASE function
examples of, 336
syntax diagram of, 336
Union operation
associative, UNION ALL, 85
columns, characteristics of, 85
ORDER BY clause restriction, 85
SELECT statement use of, 82
UNIQUE constraint, 28, 29, 38, 113
UPD_ORDERED, 353
Updatable view, requirements for, 48
UPDATE statement
authorization requirements, 97
conflicting updates, 98
description of, 96
examples of, 99
isolation levels, 97
performance, 97
SET clause, 96
syntax diagram of, 96
WHERE clause, 97
UPDATE STATISTICS statement
column groups, 101
column lists, 101
considerations, 102
examples of, 103
histogram tables, 101
syntax diagram of, 100
UPPER function
elements of, 337
syntax diagram of, 337
Upset
using single row, 68
UPsert statement
elements of, 105
syntax diagram of, 105
UPSHIFT function
syntax diagram of, 338
V
Value expressions, 129
summary of, 200
CASE (Conditional) expression, 211
CAST expression, 214
VALUES statement
considerations for, 107
examples, 107
syntax diagram for, 107
VARCHAR data type, 122, 123
Variable-length character column, 123
VARIANCE function
DISTINCT clause within, 339
elements of, 340
statistical definition of, 339
syntax diagram of, 339
Vertical partition
example, 50
Views
CREATE VIEW statement, 47
description of, 171
DROP VIEW statement, 55
relationship to tables, 170
upatability requirements, 48
Volatile tables
considerations, 40, 41
examples, 42
nullable constraints, 42
nullable keys, 41
nullable primary key, 42
suitable keys, 41
W
WEEK function
example of, 341
syntax diagram of, 341
Y
YEAR function
elements of, 342
syntax diagram of, 342
Z
ZEROIFNULL function
example of, 343
syntax diagram of, 343