Red Hat Integration 2020.Q1

Data Virtualization Reference

TECHNOLOGY PREVIEW - Reference for Data Virtualization

Last Updated: 2021-02-19

Red Hat Integration 2020.Q1 Data Virtualization Reference

TECHNOLOGY PREVIEW - Reference for Data Virtualization

Legal Notice

Copyright © 2021 Red Hat, Inc.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at
http://creativecommons.org/licenses/by-sa/3.0/
. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, the Red Hat logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux ® is the registered trademark of Linus Torvalds in the United States and other countries.

Java ® is a registered trademark of Oracle and/or its affiliates.

XFS ® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL ® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js ® is an official trademark of Joyent. Red Hat is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack ® Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

Abstract

General reference information for Data Virtualization.

Table of Contents

CHAPTER 1. DATA VIRTUALIZATION REFERENCE **10**

CHAPTER 2. VIRTUAL DATABASES **11**
2.1. VIRTUAL DATABASE PROPERTIES 12
2.2. DDL METADATA FOR SCHEMA OBJECTS 14
2.3. DDL METADATA FOR DOMAINS 28

CHAPTER 3. SQL COMPATIBILITY **29**
3.1. IDENTIFIERS 29
3.2. OPERATOR PRECEDENCE 30
3.3. EXPRESSIONS 30
3.3.1. Column Identifiers 31
3.3.2. Literals 31
3.3.3. Window functions 35
3.3.4. Case and searched case 38
3.3.5. Scalar subqueries 38
3.3.6. Parameter references 38
3.3.7. Arrays 38
3.4. CRITERIA 39
3.5. SCALAR FUNCTIONS 42
3.5.1. Numeric functions 42
3.5.2. String functions 46
3.5.3. Date and time functions 50
3.5.4. Type conversion functions 56
3.5.5. Choice functions 57
3.5.6. Decode functions 57
3.5.7. Lookup function 59
3.5.8. System functions 59
3.5.9. XML functions 62
3.5.10. JSON functions 69
3.5.11. Security functions 74
3.5.12. Spatial functions 76
3.5.13. Miscellaneous functions 87
3.5.14. Nondeterministic function handling 89
3.6. DML COMMANDS 90
3.6.1. Set operations 91
3.6.2. SELECT command 91
3.6.3. VALUES command 92
3.6.4. Update commands 93
3.6.4.1. INSERT command 93
3.6.4.2. UPDATE command 93
3.6.4.3. DELETE command 93
3.6.4.4. UPSERT (MERGE) command 93
3.6.4.5. EXECUTE command 94
3.6.4.6. Procedural relational command 94
3.6.4.7. Anonymous procedure block 95
3.6.5. Subqueries 96
3.6.6. WITH clause 97
3.6.7. SELECT clause 98
3.6.8. FROM clause 98
3.6.8.1. Nested tables 100

3.6.8.2. XMLTABLE 101
3.6.8.3. ARRAYTABLE 102
3.6.8.4. OBJECTTABLE 103
3.6.8.5. TEXTTABLE 105
3.6.8.6. JSONTABLE 107
3.6.9. WHERE clause 108
3.6.10. GROUP BY clause 109
3.6.11. HAVING Clause 110
3.6.12. ORDER BY clause 110
3.6.13. LIMIT clause 111
3.6.14. INTO clause 112
3.6.15. OPTION clause 113
3.7. DDL COMMANDS 114
3.7.1. Temporary Tables 114
3.7.1.1. Local temporary tables 114
3.7.1.2. Global temporary tables 115
3.7.1.3. Common features of global and local temporary tables 116
3.7.1.4. Foreign temporary tables 116
3.7.2. Alter view 118
3.7.3. Alter procedure 118
3.7.4. Alter trigger 118
3.8. PROCEDURES 118
3.8.1. Procedure language 118
3.8.1.1. Command statement 119
3.8.1.2. Dynamic SQL command 120
3.8.1.3. Declaration statement 122
3.8.1.4. Assignment statement 123
3.8.1.5. Special variables 123
3.8.1.6. Compound statement 124
3.8.1.7. IF statement 126
3.8.1.8. Loop Statement 126
3.8.1.9. While statement 127
3.8.1.10. Continue statement 127
3.8.1.11. Break statement 127
3.8.1.12. Leave statement 128
3.8.1.13. Return statement 128
3.8.1.14. Error statement 128
3.8.1.15. Raise statement 129
3.8.1.16. Exception expression 129
3.8.2. Virtual procedures 129
3.8.3. Triggers 132
3.9. COMMENTS 135
3.10. EXPLAIN STATEMENTS 135

CHAPTER 4. DATA TYPES **137**
4.1. RUNTIME TYPES 137
4.2. TYPE CONVERSIONS 140
4.3. SPECIAL CONVERSION CASES 143
4.4. ESCAPED LITERAL SYNTAX 144

CHAPTER 5. UPDATABLE VIEWS **145**
5.1. KEY-PRESERVED TABLES 146

CHAPTER 6. TRANSACTIONS **147**

6.1. AUTOCOMMITTXN EXECUTION PROPERTY 147
6.2. UPDATING MODEL COUNT 148
6.3. JDBC AND TRANSACTIONS 148
6.4. LIMITATIONS 149

CHAPTER 7. DATA ROLES **150**
7.1. PERMISSIONS 150
7.2. ROLE MAPPING 154

CHAPTER 8. SYSTEM SCHEMA **155**
8.1. SYS SCHEMA 155
8.2. SYSADMIN SCHEMA 167
8.2.1. SYSADMIN.refreshMatView 173
8.2.2. SYSADMIN.refreshMatViewRow 173
8.2.3. SYSADMIN.refreshMatViewRows 174
8.2.4. SYSADMIN.setColumnStats 174
8.2.5. SYSADMIN.setProperty 175
8.2.6. SYSADMIN.setTableStats 175

CHAPTER 9. TRANSLATORS **178**
9.1. AMAZON S3 TRANSLATOR 181
9.2. DELEGATOR TRANSLATORS 186
9.2.1. Extending the delegator translator 186
9.3. FILE TRANSLATOR 188
9.4. GOOGLE SPREADSHEET TRANSLATOR 190
9.5. JDBC TRANSLATORS 191
9.5.1. Actian Vector translator (actian-vector) 200
9.5.2. Apache Phoenix Translator (phoenix) 201
9.5.3. Cloudera Impala translator (impala) 201
9.5.4. Db2 Translator (db2) 202
9.5.5. Derby translator (derby) 203
9.5.6. Exasol translator (exasol) 203
9.5.7. Greenplum Translator (greenplum) 203
9.5.8. H2 Translator (h2) 203
9.5.9. Hive Translator (hive) 203
9.5.10. HSQL Translator (hsql) 204
9.5.11. Informix translator (informix) 204
9.5.12. Ingres translators (ingres / ingres93) 205
9.5.13. Intersystems Caché translator (intersystems-cache) 205
9.5.14. JDBC ANSI translator (jdbc-ansi) 205
9.5.15. JDBC simple translator (jdbc-simple) 205
9.5.16. Microsoft Access translators 205
9.5.17. Microsoft SQL Server translator (sqlserver) 206
9.5.18. MySQL translator (mysql/mysql5) 206
9.5.19. Netezza translator (netezza) 207
9.5.20. Oracle translator (oracle) 207
9.5.21. PostgreSQL translator (postgresql) 210
9.5.22. PrestoDB translator (prestodb) 210
9.5.23. Redshift translator (redshift) 211
9.5.24. SAP HANA translator (hana) 212
9.5.25. SAP IQ translator (sap-iq) 212
9.5.26. Sybase translator (sybase) 212
9.5.27. Data Virtualization translator (teiid) 213
9.5.28. Teradata translator (teradata) 213

9.5.29. Vertica translator (vertica) 213
9.6. LOOPBACK TRANSLATOR 213
9.7. MICROSOFT EXCEL TRANSLATOR 214
9.8. MONGODB TRANSLATOR 217
9.9. ODATA TRANSLATOR 230
9.10. ODATA V4 TRANSLATOR 232
9.11. OPENAPI TRANSLATOR 235
9.12. SALESFORCE TRANSLATORS 237
9.13. REST TRANSLATOR 245
9.14. WEB SERVICES TRANSLATOR 247

CHAPTER 10. FEDERATED PLANNING **250**
10.1. PLANNING OVERVIEW 250
10.2. QUERY PLANNER 251
10.3. QUERY PLANS 264
10.4. FEDERATED OPTIMIZATIONS 273
10.5. SUBQUERY OPTIMIZATION 278
10.6. XQUERY OPTIMIZATION 279
10.7. FEDERATED FAILURE MODES 280
10.8. CONFORMED TABLES 280

CHAPTER 11. DATA VIRTUALIZATION ARCHITECTURE **282**
11.1. TERMINOLOGY 283
11.2. DATA MANAGEMENT 284
11.3. QUERY TERMINATION 285
11.4. PROCESSING 285

CHAPTER 12. BNF FOR SQL GRAMMAR **287**
12.1. RESERVED KEYWORDS 287
12.2. NON-RESERVED KEYWORDS 297
12.3. RESERVED KEYWORDS FOR FUTURE USE 303
12.4. TOKENS 304
12.5. PRODUCTION CROSS-REFERENCE 309
12.6. PRODUCTIONS 320
12.6.1. string ::= 320
12.6.2. non-reserved identifier ::= 320
12.6.3. basicNonReserved ::= 320
12.6.4. Unqualified identifier ::= 324
12.6.5. identifier ::= 324
12.6.6. create trigger ::= 325
12.6.7. alter ::= 325
12.6.8. for each row trigger action ::= 325
12.6.9. explain ::= 325
12.6.10. explain option ::= 326
12.6.11. directly executable statement ::= 326
12.6.12. drop table ::= 326
12.6.13. create temporary table ::= 326
12.6.14. temporary table element ::= 327
12.6.15. raise error statement ::= 327
12.6.16. raise statement ::= 327
12.6.17. exception reference ::= 327
12.6.18. sql exception ::= 328
12.6.19. statement ::= 328
12.6.20. delimited statement ::= 328

12.6.21. compound statement ::= 328
12.6.22. branching statement ::= 328
12.6.23. return statement ::= 329
12.6.24. while statement ::= 329
12.6.25. loop statement ::= 329
12.6.26. if statement ::= 329
12.6.27. declare statement ::= 329
12.6.28. assignment statement ::= 330
12.6.29. assignment statement operand ::= 330
12.6.30. data statement ::= 330
12.6.31. dynamic data statement ::= 330
12.6.32. set clause list ::= 330
12.6.33. typed element list ::= 331
12.6.34. callable statement ::= 331
12.6.35. call statement ::= 331
12.6.36. named parameter list ::= 331
12.6.37. insert statement ::= 331
12.6.38. expression list ::= 332
12.6.39. update statement ::= 332
12.6.40. delete statement ::= 332
12.6.41. query expression ::= 332
12.6.42. with list element ::= 332
12.6.43. query expression body ::= 333
12.6.44. query term ::= 333
12.6.45. query primary ::= 333
12.6.46. query ::= 333
12.6.47. into clause ::= 334
12.6.48. select clause ::= 334
12.6.49. select sublist ::= 334
12.6.50. select derived column ::= 334
12.6.51. derived column ::= 335
12.6.52. all in group ::= 335
12.6.53. ordered aggregate function ::= 335
12.6.54. text aggreate function ::= 335
12.6.55. standard aggregate function ::= 335
12.6.56. analytic aggregate function ::= 336
12.6.57. filter clause ::= 336
12.6.58. from clause ::= 336
12.6.59. table reference ::= 336
12.6.60. joined table ::= 337
12.6.61. cross join ::= 337
12.6.62. qualified table ::= 337
12.6.63. table primary ::= 337
12.6.64. make dep options ::= 337
12.6.65. xml serialize ::= 338
12.6.66. array table ::= 338
12.6.67. json table ::= 338
12.6.68. json table column ::= 338
12.6.69. text table ::= 338
12.6.70. text table column ::= 339
12.6.71. xml query ::= 339
12.6.72. xml query ::= 339
12.6.73. object table ::= 339

12.6.74. object table column ::= 340
12.6.75. xml table ::= 340
12.6.76. xml table column ::= 340
12.6.77. unsigned integer ::= 340
12.6.78. table subquery ::= 340
12.6.79. table name ::= 341
12.6.80. where clause ::= 341
12.6.81. condition ::= 341
12.6.82. boolean value expression ::= 341
12.6.83. boolean term ::= 341
12.6.84. boolean factor ::= 341
12.6.85. boolean primary ::= 342
12.6.86. comparison operator ::= 342
12.6.87. is distinct ::= 342
12.6.88. comparison predicate ::= 342
12.6.89. subquery ::= 343
12.6.90. quantified comparison predicate ::= 343
12.6.91. match predicate ::= 343
12.6.92. like regex predicate ::= 343
12.6.93. character ::= 343
12.6.94. between predicate ::= 344
12.6.95. is null predicate ::= 344
12.6.96. in predicate ::= 344
12.6.97. exists predicate ::= 344
12.6.98. group by clause ::= 344
12.6.99. having clause ::= 345
12.6.100. order by clause ::= 345
12.6.101. sort specification ::= 345
12.6.102. sort key ::= 345
12.6.103. integer parameter ::= 345
12.6.104. limit clause ::= 346
12.6.105. fetch clause ::= 346
12.6.106. option clause ::= 346
12.6.107. expression ::= 346
12.6.108. common value expression ::= 347
12.6.109. numeric value expression ::= 347
12.6.110. plus or minus ::= 347
12.6.111. term ::= 347
12.6.112. star or slash ::= 347
12.6.113. value expression primary ::= 348
12.6.114. parameter reference ::= 348
12.6.115. unescapedFunction ::= 348
12.6.116. nested expression ::= 348
12.6.117. unsigned value expression primary ::= 348
12.6.118. ARRAY expression constructor ::= 349
12.6.119. window specification ::= 349
12.6.120. window frame ::= 349
12.6.121. window frame bound ::= 349
12.6.122. case expression ::= 350
12.6.123. searched case expression ::= 350
12.6.124. function ::= 350
12.6.125. xml parse ::= 351
12.6.126. querystring function ::= 351

12.6.127. xml element ::= 351
12.6.128. xml attributes ::= 352
12.6.129. json object ::= 352
12.6.130. derived column list ::= 352
12.6.131. xml forest ::= 352
12.6.132. xml namespaces ::= 352
12.6.133. xml namespace element ::= 353
12.6.134. simple data type ::= 353
12.6.135. basic data type ::= 354
12.6.136. data type ::= 354
12.6.137. time interval ::= 354
12.6.138. non numeric literal ::= 355
12.6.139. unsigned numeric literal ::= 355
12.6.140. ddl statement ::= 356
12.6.141. option namespace ::= 357
12.6.142. create database ::= 357
12.6.143. use database ::= 357
12.6.144. create schema ::= 357
12.6.145. drop schema ::= 357
12.6.146. set schema ::= 358
12.6.147. create a domain or type alias ::= 358
12.6.148. create data wrapper ::= 358
12.6.149. Drop data wrapper ::= 358
12.6.150. create role ::= 358
12.6.151. with role ::= 359
12.6.152. drop role ::= 359
12.6.153. CREATE POLICY ::= 359
12.6.154. DROP POLICY ::= 359
12.6.155. GRANT ::= 359
12.6.156. Revoke GRANT ::= 360
12.6.157. create server ::= 360
12.6.158. drop server ::= 360
12.6.159. create procedure ::= 360
12.6.160. drop procedure ::= 361
12.6.161. procedure parameter ::= 361
12.6.162. procedure result column ::= 361
12.6.163. create table ::= 361
12.6.164. create foreign or global temporary table ::= 361
12.6.165. create view ::= 362
12.6.166. drop table ::= 362
12.6.167. create foreign temp table ::= 362
12.6.168. create table body ::= 362
12.6.169. create view body ::= 362
12.6.170. table constraint ::= 363
12.6.171. foreign key ::= 363
12.6.172. primary key ::= 363
12.6.173. other constraints ::= 363
12.6.174. column list ::= 363
12.6.175. table element ::= 364
12.6.176. view element ::= 364
12.6.177. post create column ::= 364
12.6.178. inline constraint ::= 364
12.6.179. options clause ::= 364

12.6.180. option pair ::= 365
12.6.181. alter option pair ::= 365
12.6.182. alterStatement ::= 365
12.6.183. ALTER TABLE ::= 365
12.6.184. RENAME Table ::= 365
12.6.185. ADD constraint ::= 366
12.6.186. ADD column ::= 366
12.6.187. DROP column ::= 366
12.6.188. alter column options ::= 366
12.6.189. rename column options ::= 366
12.6.190. ALTER PROCEDURE ::= 367
12.6.191. ALTER TRIGGER ::= 367
12.6.192. ALTER SERVER ::= 367
12.6.193. ALTER DATA WRAPPER ::= 367
12.6.194. ALTER DATABASE ::= 367
12.6.195. alter options list ::= 368
12.6.196. drop option ::= 368
12.6.197. add set option ::= 368
12.6.198. alter child options list ::= 368
12.6.199. drop option ::= 368
12.6.200. add set child option ::= 369
12.6.201. alter child option pair ::= 369
12.6.202. Import foreign schema ::= 369
12.6.203. Import another Database ::= 369
12.6.204. identifier list ::= 369
12.6.205. grant type ::= 369

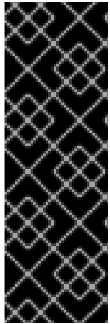
CHAPTER 1. DATA VIRTUALIZATION REFERENCE

Data Virtualization offers a highly scalable and high performance solution to information integration. By allowing integrated and enriched data to be consumed relationally, as JSON, XML, and other formats over multiple protocols. Data Virtualization simplifies data access for developers and consuming applications.

Commercial development support, production support, and training for Data Virtualization is available through Red Hat. Data Virtualization is a professional open source project and a critical component of Red Hat data Integration.

Before one can delve into Data Virtualization it is very important to learn few basic constructs of Data Virtualization. For example, what is a virtual database? What is a model? and so forth. For more information, see the Teiid Basics.

If not otherwise specified, versions referenced in this document refer to Teiid project versions. Teiid or Data Virtualization running on various platforms will have both platform and product-specific versioning.

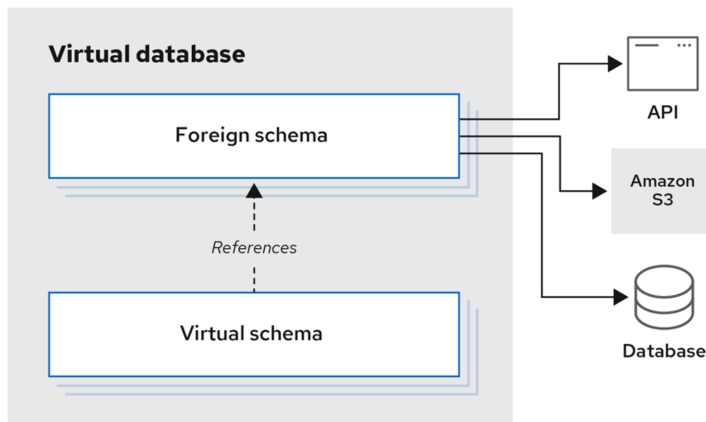


IMPORTANT

Data virtualization is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process. For more information about the support scope of Red Hat Technology Preview features, see https://access.redhat.com/support/offerings/techpreview/.

CHAPTER 2. VIRTUAL DATABASES

A virtual database (VDB) is a metadata container for components used to integrate data from multiple data sources, so that they can be accessed in an integrated manner through a single, uniform API.



A virtual database typically contains multiple schema components (also called as models), and each schema contains the metadata (tables, procedures, functions). There are two different types of schemas:

Foreign schema

Also called a *source* or *physical* schema, a foreign schema represents external or remote data sources, such as a relational database, such as Oracle, Db2, or MySQL; files, such as CSV or Microsoft Excel; or web services, such as SOAP or REST.

Virtual schema

A view layer, or logical schema layer that is defined using schema objects from *foreign schemas*. For example, when you create a view table that aggregates multiple foreign tables from different sources, the resulting view shields users from the complexities of the data sources that define the view.

One important thing to note is, a virtual database contains only metadata. Any use case involving Data Virtualization must have a virtual database model to begin with. So, it is important to learn how to design and develop a VDB.

The following example of a virtual database model, defines a single foreign schema component that makes a connection to a PostgreSQL database.

The SQL DDL commands in the example implement the SQL/MED specification.

```
CREATE DATABASE my_example;
USE DATABASE my_example;
CREATE SERVER pgsql
  VERSION 'one' FOREIGN DATA WRAPPER postgresql
  OPTIONS (
    "resource-name" 'java:/postgres-ds'
  );
CREATE SCHEMA test SERVER pgsql;
IMPORT FOREIGN SCHEMA public FROM SERVER pgsql INTO test
  OPTIONS(
```

```
        importer.useFullSchemaName false,
        importer.tableTypes 'TABLE,VIEW'
);
```

The following sections describe in greater detail how the statements in the preceding example are used to define a virtual database. Before that we need to learn about the different elements of the *source schema* component.

External data sources

As shown in preceding example, the "source schema" component of a virtual database is a collection of schema objects, tables, procedures and functions, that represent an external data source’s metadata locally. In the example, schema objects are not defined directly, but are imported from the server. Details of the connection to the external data source are provided through a **resource-name**, which is a named connection reference to a external data source.

For the purposes of Data Virtualization, connecting and issuing queries to fetch the metadata from these external data sources, Data Virtualization defines/provides two types of resources.

Translator

A translator, also known as a **DATA WRAPPER**, is a component that provides an abstraction layer between the Data Virtualization query engine and a physical data source. The translator knows how to convert query commands from Data Virtualization into source-specific commands and execute them. The translator also has the intelligence to convert data that the physical source returns into a form that the Data Virtualization query engine can process. For example, when working with a web service translator, the translator converts SQL procedures from the Data Virtualization layer into HTTP calls, and JSON responses are converted to tabular results.

Data Virtualization provides various translators as part of the system, or one can be developed by using the provided java libraries. For information about the available translators, see Translators.

2.1. VIRTUAL DATABASE PROPERTIES

DATABASE properties

- *domain-ddl*
- *schema-ddl*
- *query-timeout* Sets the default query timeout in milliseconds for queries executed against this VDB. **0** indicates that the server default query timeout should be used. Defaults to 0. Will have no effect if the server default query timeout is set to a lesser value. Note that clients can still set their own timeouts that will be managed on the client side.
- *connection.XXX* For use by the ODBC transport and OData to set default connection/execution properties. For more information about related properties, see *Driver Connection* in the Client Developer’s Guide. Note these are set on the connection after it has been established.

```
CREATE DATABASE vdb OPTIONS ("connection.partialResultsMode" true);
```

- *authentication-type*

Authentication type of configured security domain. Allowed values currently are (GSS, USERPASSWORD). The default is set on the transport (typically USERPASSWORD).

- *password-pattern*

Regular expression matched against the connecting user's name that determines if USERPASSWORD authentication is used. *password-pattern* takes precedence over *authentication-type*. The default is *authentication-type*.

- *gss-pattern*

Regular expression matched against the connecting user's name that determines if GSS authentication is used. *gss-pattern* takes precedence over *password-pattern*. The default is *password-pattern*.

- *max-sessions-per-user* (11.2+)

Maximum number of sessions allowed for each user, as identified by the user name, of this VDB. No setting or a negative number indicates no per user max, but the session service max will still apply. This is enforced at each Data Virtualization server member in a cluster, and not cluster wide. Derived sessions that are created for tasks under an existing session do not count against this maximum.

- *model.visible*

Used to override the visibility of imported vdb models, where model is the name of the imported model.

- *include-pg-metadata*

By default, PostgreSQL metadata is always added to VDB unless you set the property *org.teiid.addPGMetadata* to false. This property enables adding PG metadata per VDB. For more information, *System Properties* in the Administrator's Guide. Please note that if you are using ODBC to access your VDB, the VDB must include PG metadata.

- *lazy-invalidate*

By default TTL expiration will be invalidating. For more information, see *Internal Materialization* in the Caching guide. Setting lazy-invalidate to **true** will make TTL refreshes non-invalidating.

- *deployment-name*

Effectively reserved. Will be set at deploy time by the server to the name of the server deployment.

Schema and model properties

- *visible*

Marks the schema as visible when the value is **true** (the default setting). When the **visible** flag is set to **false**, the schema's metadata is hidden from any metadata requests. Setting the property to **false** does not prohibit you from issuing queries against this schema. For information about how to control access to data, see Data roles.

- *multisource*

Sets the schema to multi-source mode, where the data exists in partitions in multiple different sources. It is assumed that metadata of the schema is the same across all data sources.

- *multisource.columnName*

In a multi-source schema, an additional column that designates the partition is implicitly added to all tables to identify the source. This property defines the name of that column, the type will be always **String**.

- *multisource.addColumn*

This flag specifies to add an implicit partition column to all the tables in this schema. A *true* value adds the column. Default is *false*.

- *allowed-languages*

Specifies a comma-separated list of programming languages that can be used for any purpose in the VDB. Names are case-sensitive, and the list cannot include whitespace between entries. For example, **<property name="allowed-languages" value="javascript"/>**

- *allow-language* Specifies that a role has permission to use a language that is listed in the **allowed-languages** property. For example, the **allow-language** property in following excerpt specifies that users with the role **RoleA** have permission to use Javascript.

```
<data-role name="RoleA">
    <description>Read and javascript access.</description>

    <permission>
      <resource-name>modelName</resource-name>
      <allow-read>true</allow-read>
    </permission>

    <permission>
      <resource-name>javascript</resource-name>
      <allow-language>true</allow-language>
    </permission>

    <mapped-role-name>role1</mapped-role-name>

  </data-role>
```

2.2. DDL METADATA FOR SCHEMA OBJECTS

Tables and views exist in the same namespace in a schema. Indexes are not considered schema scoped objects, but are rather scoped to the table or view they are defined against. Procedures and functions are defined in separate namespaces, but a function that is defined by virtual procedure language exists as both a function and a procedure of the same name. Domain types are not schema-scoped; they are scoped to the entire VDB.

Data types

For information about data types, see *simple data type* in the BNF for SQL grammar.

Foreign tables

A *FOREIGN* table is table that is defined on source schema that represents a real relational table in source databases such as Oracle, Microsoft SQL Server, and so forth. For relational databases, Data Virtualization can automatically retrieve the database schema information upon the deployment of the VDB, if you want to auto import the existing schema. However, users can use the following FOREIGN table semantics, when they would like to explicitly define tables on PHYSICAL schema or represent non-relational data as relational in custom translators.

Example: Create foreign table (Created on PHYSICAL model)

```
CREATE FOREIGN TABLE {table-name} (
```

```
    <table-element> (,<table-element>)*
    (,<constraint>)*
) [OPTIONS (<options-clause>)]


<table-element> ::=
    {column-name} <data-type> <element-attr> <options-clause>

<data-type> ::=
    varchar | boolean | integer | double | date | timestamp .. (see Data Types)

<element-attr> ::=
    [AUTO_INCREMENT] [NOT NULL] [PRIMARY KEY] [UNIQUE] [INDEX] [DEFAULT {expr}]

<constraint> ::=
    CONSTRAINT {constraint-name} (
        PRIMARY KEY <columns> |
        FOREIGN KEY (<columns>) REFERENCES tbl (<columns>)
        UNIQUE <columns> |
        ACCESSPATTERN <columns>
        INDEX <columns>

<columns> ::=
    ( {column-name} [,{column-name}]* )

<options-clause> ::=
    <key> <value>[,<key>, <value>]*
```

For more information about creating foreign tables, see *CREATE TABLE* in BNF for SQL grammar.

Example: Create foreign table (Created on PHYSICAL model)

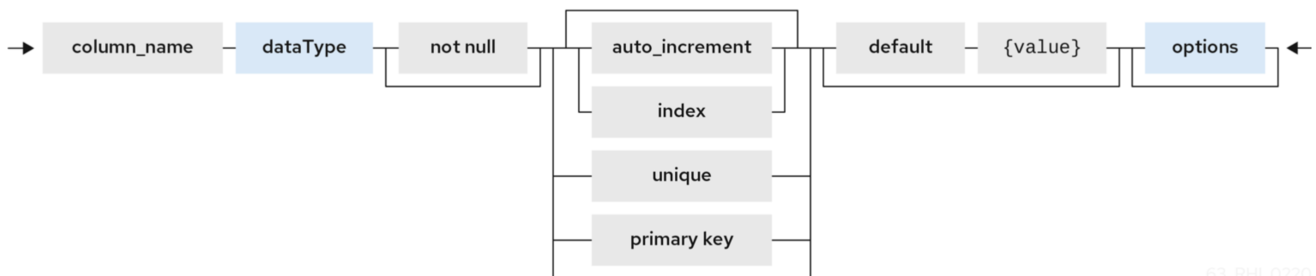
```
CREATE FOREIGN TABLE Customer (
    id integer PRIMARY KEY,
    firstname varchar(25),
    lastname varchar(25),
    dob timestamp);

CREATE FOREIGN TABLE Order (
    id integer PRIMARY KEY,
    customerid integer OPTIONS(ANNOTATION 'Customer primary key'),
    saledate date,
    amount decimal(25,4),
    CONSTRAINT CUSTOMER_FK FOREIGN KEY(customerid) REFERENCES Customer(id)
 ) OPTIONS(UPDATABLE true, ANNOTATION 'Orders Table');
```

TABLE OPTIONS: (the following options are well known, any others properties defined will be considered as extension metadata)

Property	Data type or allowed values	Description
UUID	string	Unique identifier for the view.

Property	Data type or allowed values	Description
CARDINALITY	int	Costing information. Number of rows in the table. Used for planning purposes.
UPDATABLE	'TRUE'	'FALSE'
Defines whether or not the view is allowed to update.	ANNOTATION	string
Description of the view.	DETERMINISM	NONDETERMINISTIC, COMMAND_DETERMINISTIC, SESSION_DETERMINISTIC, USER_DETERMINISTIC, VDB_DETERMINISTIC, DETERMINISTIC



COLUMN OPTIONS: (the following options are well known, any others properties defined will be considered as extension metadata).

Property	Data type or allowed values	Description
UUID	string	A unique identifier for the column.
NAMEINSOURCE	string	If this is a column name on the FOREIGN table, this value represents name of the column in source database. If omitted, the column name is used when querying for data against the source.
CASE_SENSITIVE	'TRUE'|'FALSE'	
SELECTABLE	'TRUE'|'FALSE'	TRUE when this column is available for selection from the user query.

Property	Data type or allowed values	Description
UPDATABLE	'TRUE'|'FALSE'	Defines if the column is updatable. Defaults to true if the view/table is updatable.
SIGNED	'TRUE'|'FALSE'	
CURRENCY	'TRUE'|'FALSE'	
FIXED_LENGTH	'TRUE'|'FALSE'	
SEARCHABLE	'SEARCHABLE'|'UNSEARCHABLE'|'LIKE_ONLY'|'ALL_EXCEPT_LIKE'	Column searchability. Usually dictated by the data type.
MIN_VALUE		
MAX_VALUE		
CHAR_OCTET_LENGTH	integer	
ANNOTATION	string	
NATIVE_TYPE	string	
RADIX	integer	
NULL_VALUE_COUNT	long	Costing information. Number of NULLS in this column.
DISTINCT_VALUES	long	Costing information. Number of distinct values in this column.

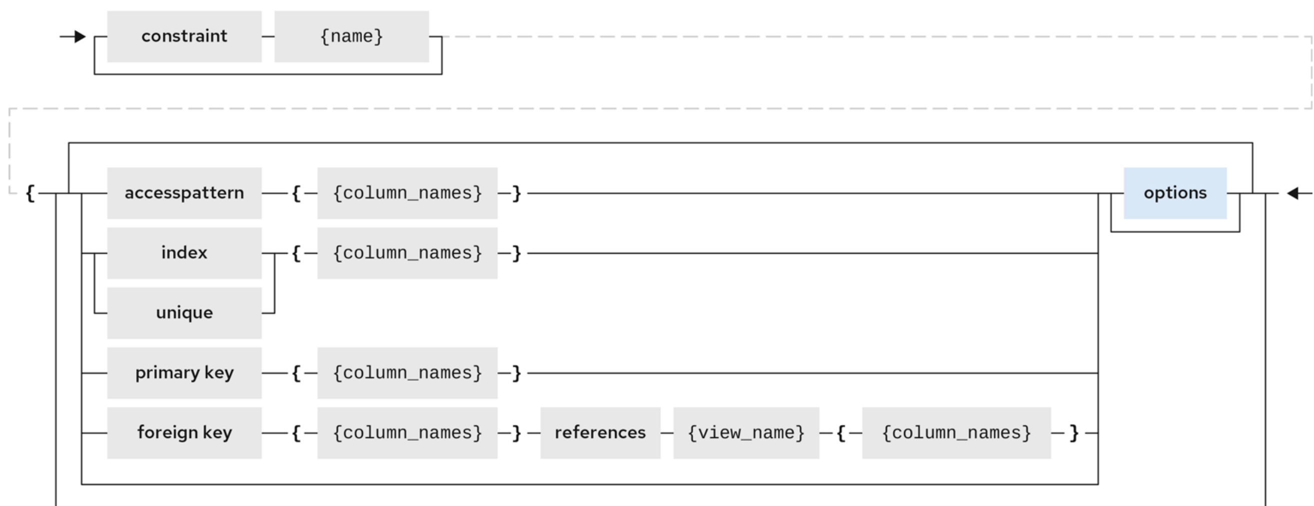
Columns may also be marked as NOT NULL, auto_increment, or with a DEFAULT value.

A column of type bigdecimal/decimal/numeric can be declared without a precision/scale, which defaults to an internal maximum for precision with half scale, or with a precision which will default to a scale of 0.

A column of type timestamp can be declared without a scale which will default to an internal maximum of 9 fractional seconds.

Table Constraints

Constraints can be defined on table/view to define indexes and relationships to other tables/views. This information is used by the Data Virtualization optimizer to plan queries, or use the indexes in materialization tables to optimize the access to the data.



CONSTRAINTS are same as one can define on RDBMS.

Example of CONSTRAINTs

```
CREATE FOREIGN TABLE Orders (
    name varchar(50),
    saledate date,
    amount decimal,
    CONSTRAINT CUSTOMER_FK FOREIGN KEY(customerid) REFERENCES Customer(id)
    ACCESSPATTERN (name),
    PRIMARY KEY ...
    UNIQUE ...
    INDEX ...
```

ALTER TABLE

For the full SQL grammar for the ALTER TABLE statement, see *ALTER TABLE* in the BNF for SQL grammar.

Using the ALTER command, one can Add, Change, Delete columns, modify the values of any OPTIONS, and add constraints. The following examples show how to use the ALTER command to modify table objects.

```
-- add column to the table
ALTER FOREIGN TABLE "Customer" ADD COLUMN address varchar(50) OPTIONS(SELECTABLE
true);

-- remove column to the table
ALTER FOREIGN TABLE "Customer" DROP COLUMN address;

-- adding options property on the table
ALTER FOREIGN TABLE "Customer" OPTIONS (ADD CARDINALITY 10000);

-- Changing options property on the table
ALTER FOREIGN TABLE "Customer" OPTIONS (SET CARDINALITY 9999);

-- Changing options property on the table's column
ALTER FOREIGN TABLE "Customer" ALTER COLUMN "name" OPTIONS(SET UPDATABLE
```

```
FALSE)

-- Changing table's column type to integer
ALTER FOREIGN TABLE "Customer" ALTER COLUMN "id" TYPE bigdecimal;

-- Changing table's column column name
ALTER FOREIGN TABLE "Customer" RENAME COLUMN "id" TO "customer_id";

-- Adding a constraint
ALTER VIEW "Customer_View" ADD PRIMARY KEY (id);
```

Views

A view is a virtual table. A view contains rows and columns, like a real table. The columns in a view are columns from one or more real tables from the source or other view models. They can also be expressions made up multiple columns, or aggregated columns. When column definitions are not defined on the view table, they are derived from the projected columns of the view's select transformation that is defined after the **AS** keyword.

You can add functions, JOIN statements and WHERE clauses to a view data as if the data were coming from one single table.

Access patterns are not currently meaningful to views, but are still allowed by the grammar. Other constraints on views are also not enforced, unless they are specified on an internal materialized view, in which case they will be automatically added to the materialization target table. However, non-access pattern View constraints are still useful for other purposes, such as to convey relationships for optimization and for discovery by clients.

BNF for CREATE VIEW

```
CREATE VIEW {table-name} [(
   <view-element> (,<view-element>)*
   (,<constraint>)*
)] [OPTIONS (<options-clause>)]
   AS {transformation_query}


<table-element> ::=
   {column-name} [<data-type> <element-attr> <options-clause>]

<data-type> ::=
   varchar | boolean | integer | double | date | timestamp .. (see Data Types)

<element-attr> ::=
   [AUTO_INCREMENT] [NOT NULL] [PRIMARY KEY] [UNIQUE] [INDEX] [DEFAULT {expr}]

<constraint> ::=
   CONSTRAINT {constraint-name} (
      PRIMARY KEY <columns> |
      FOREIGN KEY (<columns>) REFERENCES tbl (<columns>)
      UNIQUE <columns> |
      ACCESSPATTERN <columns>
      INDEX <columns>

<columns> ::=
   ( {column-name} [,{column-name}]* )
```

```
<options-clause> ::=
  <key> <value>[,<key>, <value>]*
```

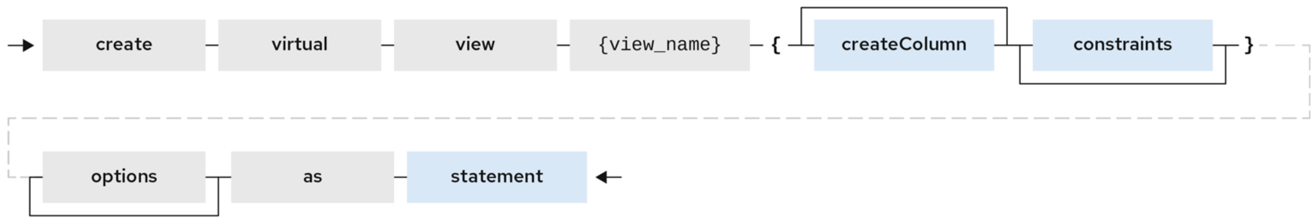
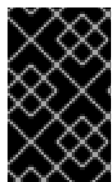


Table 2.1. VIEW OPTIONS: (These properties are in addition to properties defined in the CREATE TABLE)

Property	Data type or allowed values	Description
MATERIALIZED	'TRUE'|'FALSE'	Defines if a table is materialized.
MATERIALIZED_TABLE	'table.name'	If this view is being materialized to a external database, this defines the name of the table that is being materialized to.

Example: Create view table (created on VIRTUAL schema)

```
CREATE VIEW CustomerOrders
 AS
 SELECT concat(c.firstname, c.lastname) as name,
      o.saledate as saledate,
      o.amount as amount
 FROM Customer C JOIN Order o ON c.id = o.customerid;
```



IMPORTANT

Note that the columns are implicitly defined by the transformation query (SELECT statement). Columns can also defined inline, but if they are defined they can be only altered to modify their properties. You cannot ADD or DROP new columns.

ALTER TABLE

The BNF for ALTER VIEW, refer to ALTER TABLE

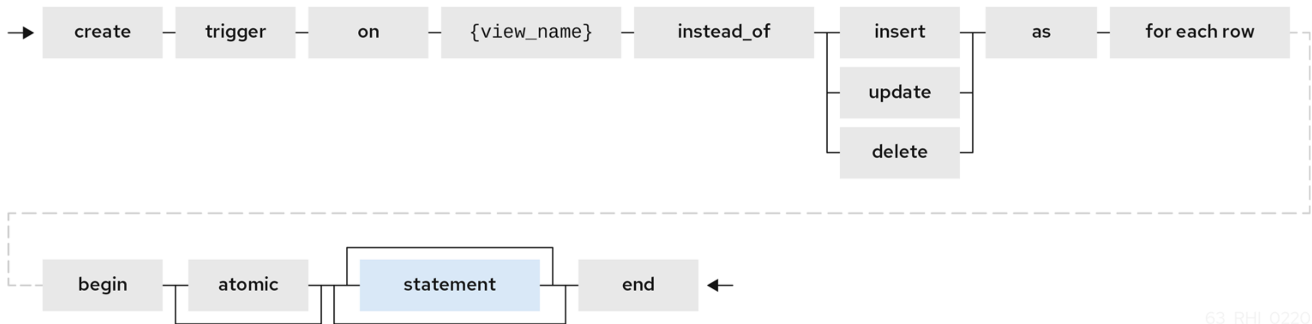
Using the ALTER COMMAND you can change the transformation query of the VIEW. You are **NOT** allowed to alter the column information. Transformation queries must be valid.

```
ALTER VIEW CustomerOrders
  AS
  SELECT concat(c.firstname, c.lastname) as name,
      o.saledate as saledate,
```

```
     o.amount as amount
FROM Customer C JOIN Order o ON c.id = o.customerid
WHERE saledate < TIMESTAMPADD(now(), -1, SQL_TSI_MONTH)
```

INSTEAD OF triggers on VIEW (Update VIEW)

A view comprising multiple base tables must use an **INSTEAD OF** trigger to insert records, apply updates, and implement deletes that reference data in the tables. Based on the select transformation's complexity some times INSTEAD OF TRIGGERS are automatically provided for the user when **UPDATABLE** OPTION on the VIEW is set to **TRUE**. However, using the CREATE TRIGGER mechanism user can provide/override the default behavior.



Example: Define INSTEAD OF trigger on View for INSERT

```
CREATE TRIGGER ON CustomerOrders INSTEAD OF INSERT AS
  FOR EACH ROW
  BEGIN ATOMIC
    INSERT INTO Customer (...) VALUES (NEW.name ...);
    INSERT INTO Orders (...) VALUES (NEW.value ...);
  END
```

For Update

Example: Define instead of trigger on View for UPDATE

```
CREATE TRIGGER ON CustomerOrders INSTEAD OF UPDATE AS
  FOR EACH ROW
  BEGIN ATOMIC
    IF (CHANGING.saledate)
    BEGIN
       UPDATE Customer SET saledate = NEW.saledate;
       UPDATE INTO Orders (...) VALUES (NEW.value ...);
    END
  END
```

While updating you have access to previous and new values of the columns. For more information about update procedures, see Update procedures.

AFTER triggers on source tables

A source table can have any number of uniquely named triggers registered to handle change events that are reported by a change data capture system.

Similar to view triggers AFTER insert provides access to new values via the NEW group, AFTER delete provides access to old values via the OLD group, and AFTER update provides access to both.

Example:Define AFTER trigger on Customer

```
CREATE TRIGGER ON Customer AFTER INSERT AS
  FOR EACH ROW
  BEGIN ATOMIC
    INSERT INTO CustomerOrders (CustomerName, CustomerID) VALUES (NEW.Name, NEW.ID);
  END
```

You will typically define a handler for each operation - INSERT/UPDATE/DELTE.

For more detailed information about update procedures, see Update procedures

Create procedure/function

A user can define one of the following functions:

Source Procedure ("CREATE FOREIGN PROCEDURE")

A stored procedure in source.

Source Function ("CREATE FOREIGN FUNCTION")

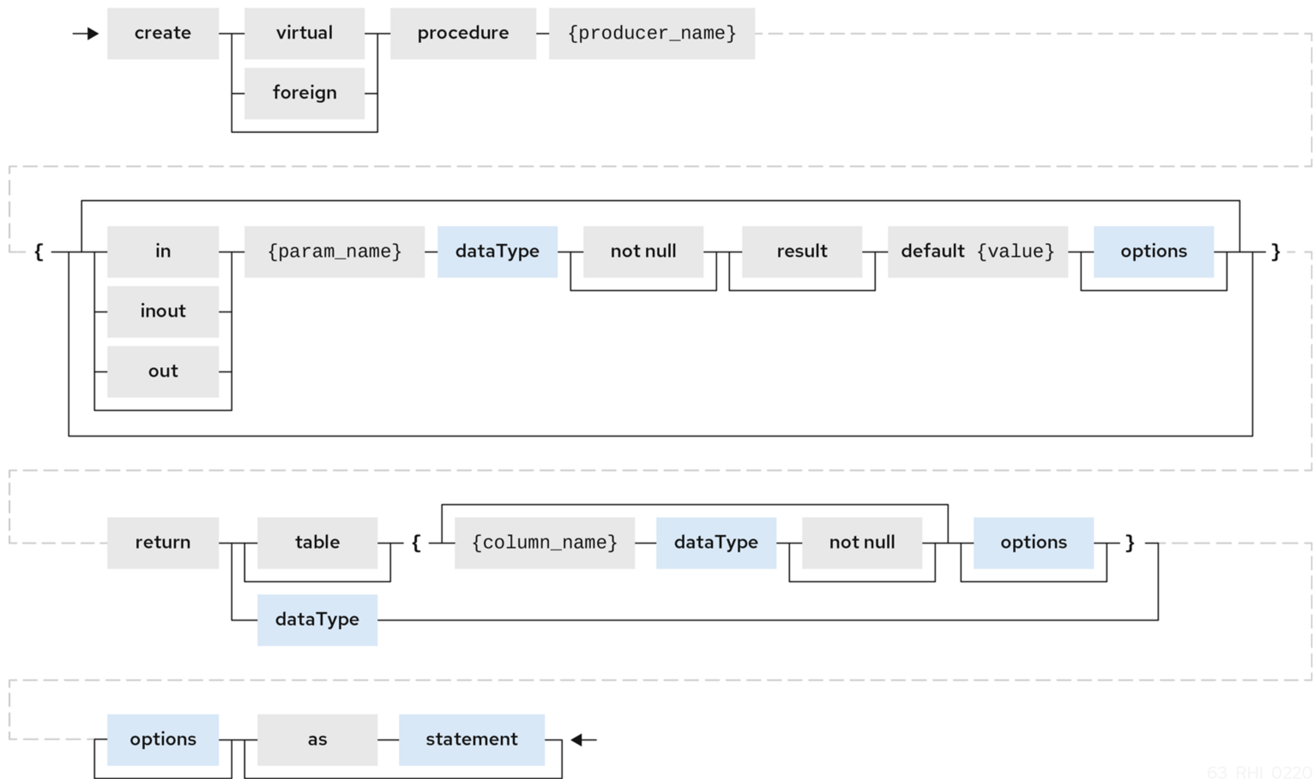
A function that depends on capabilities in the data source, and for which Data Virtualization will pushdown to the source instead of evaluating in the Data Virtualization engine.

Virtual Procedure ("CREATE VIRTUAL PROCEDURE")

Similar to stored procedure, however this is defined using the Data Virtualization’s Procedure language and evaluated in the Data Virtualization’s engine.

Function/UDF ("CREATE VIRTUAL FUNCTION")

A user defined function, that can be defined using the Teiid procedure language, or than can have the implementation defined by a Java class. For more information about writing the Java code for a UDF, see *Support for user-defined functions (non-pushdown)* in the Translator Development Guide.



For more information about creating functions or procedures, see the BNF for SQL grammar.

Variable arguments

Instead of using just an IN parameter, the last non optional parameter can be declared VARIADIC to indicate that it can be repeated 0 or more times when the procedure is called.

Example: Vararg procedure

```
CREATE FOREIGN PROCEDURE proc (x integer, VARIADIC z integer)
    RETURNS (x string);
```

FUNCTION OPTIONS:(the below are well known options, any others properties defined will be considered as extension metadata)

Property	Data Type or Allowed Values	Description
UUID	string	unique Identifier
NAMEINSOURCE	If this is source function/procedure the name in the physical source, if different from the logical name given above	
ANNOTATION	string	Description of the function/procedure
CATEGORY	string	Function Category

Property	Data Type or Allowed Values	Description
DETERMINISM	NONDETERMINISTIC, COMMAND_DETERMINISTIC, SESSION_DETERMINISTIC, USER_DETERMINISTIC, VDB_DETERMINISTIC, DETERMINISTIC	Not used on virtual procedures
NULL-ON-NULL	'TRUE'|'FALSE'	
JAVA_CLASS	string	Java Class that defines the method in case of UDF
JAVA_METHOD	string	The Java method name on the above defined java class for the UDF implementation
VARARGS	'TRUE'|'FALSE'	Indicates that the last argument of the function can be repeated 0 to any number of times. default false. It is more proper to use a VARIADIC parameter.
AGGREGATE	'TRUE'|'FALSE'	Indicates the function is a user defined aggregate function. Properties specific to aggregates are listed below.

Note that NULL-ON-NULL, VARARGS, and all of the AGGREGATE properties are also valid relational extension metadata properties that can be used on source procedures marked as functions.

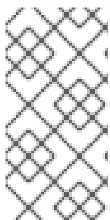
You can also create FOREIGN functions that are based on source-specific functions. For more information about creating foreign functions that use functions that are provided by the data source, see *Source supported functions* in the Translator development guide.

.AGGREGATE FUNCTION OPTIONS

Property	Data type or allowed values	Description
ANALYTIC	'TRUE'|'FALSE'	Indicates the aggregate function must be windowed. The default value is **false**.
ALLOWS-ORDERBY	'TRUE'|'FALSE'	Indicates that the aggregate function can use an ORDER BY clause. The default value is **false**.

Property	Data type or allowed values	Description
ALLOWS-DISTINCT	'TRUE'|'FALSE'	Indicates the aggregate function can use the **DISTINCT** keyword. The default value is **false**.
DECOMPOSABLE	'TRUE'|'FALSE'	Indicates the single argument aggregate function can be decomposed as agg(agg(x)) over subsets of data. The default value is **false**.
USES-DISTINCT-ROWS	'TRUE'|'FALSE'	Indicates the aggregate function effectively uses distinct rows rather than all rows. The default value is **false**.

Note that virtual functions defined using the Teiid procedure language cannot be aggregate functions.



NOTE

Providing the JAR libraries - If you have defined a UDF (virtual) function without a Teiid procedure definition, then it must be accompanied by its implementation in Java. For information about how to configure the Java library as a dependency to the VDB, see *Support for User-Defined Functions* in the Translator development guide.

PROCEDURE OPTIONS:(the following options are well known, any others properties defined will be considered as extension metadata)

Property	Data Type or Allowed Values	Description
UUID	string	Unique Identifier
NAMEINSOURCE	string	In the case of source
ANNOTATION	string	Description of the procedure
UPDATECOUNT	int	if this procedure updates the underlying sources, what is the update count, when update count is >1 the XA protocol for execution is enforced

Example: Define virtual procedure

```
CREATE VIRTUAL PROCEDURE CustomerActivity(customerid integer)
```

```
RETURNS (name varchar(25), activitydate date, amount decimal)
AS
BEGIN
  ...
END
```

For more information about virtual procedures and virtual procedure language, see Virtual procedures, and Procedure language.

Example: Define virtual function

```
CREATE VIRTUAL FUNCTION CustomerRank(customerid integer)
  RETURNS integer AS
  BEGIN
    DECLARE integer result;
    ...
    RETURN result;
  END
```

Procedure columns may also be marked as NOT NULL, or with a DEFAULT value. On a source procedure if you want the parameter to be defaultable in the source procedure and not supply a default value in Data Virtualization, then the parameter must use the extension property teiid_rel:default_handling set to omit.

There can only be a single RESULT parameter and it must be an **out** parameter. A RESULT parameter is the same as having a single non-table RETURNS type. If both are declared they are expected to match otherwise an exception is thrown. One is no more correct than the other. "RETURNS type" is shorter hand syntax especially for functions, while the parameter form is useful for additional metadata (explicit name, extension metadata, also defining a returns table, etc.).

A return parameter will be treated as the first parameter in for the procedure at runtime, regardless of where it appears in the argument list. This matches the expectation of Data Virtualization and JDBC calling semantics that expect assignments in the form "? = EXEC …".

.Relational extension OPTIONS:

Property	Data Type or Allowed Values	Description
native-query	Parameterized String	Applies to both functions and procedures. The replacement for the function syntax rather than the standard prefix form with parentheses. For more information, see *Parameterizable native queries* in Translators.
non-prepared	boolean	Applies to JDBC procedures using the native-query option. If true a PreparedStatement will not be used to execute the native query.

Example: Native query

```
CREATE FOREIGN FUNCTION func (x integer, y integer)
  RETURNS integer OPTIONS ("teiid_rel:native-query" '$1 << $2');
```

Example:Sequence native query

```
CREATE FOREIGN FUNCTION seq_nextval ()
  RETURNS integer
  OPTIONS ("teiid_rel:native-query" 'seq.nextval');
```

TIP

Use source function representations to expose sequence functionality.

Extension metadata

When defining the extension metadata in the case of Custom Translators, the properties on tables/views/procedures/columns can be whatever you need. It is recommended that you use a consistent prefix that denotes what the properties relate to. Prefixes starting with teiid_ are reserved for use by Data Virtualization. Property keys are not case sensitive when accessed via the runtime APIs - but they are case sensitive when accessing SYS.PROPERTIES.

WARNING

The usage of SET NAMESPACE for custom prefixes or namespaces is no longer allowed.

```
CREATE VIEW MyView (...)
 OPTIONS ("my-translator:mycustom-prop" 'anyvalue')
```

Table 2.2. Built-in prefixes

Prefix	Description
teiid_rel	Relational Extensions. Uses include function and native query metadata
teiid_sf	Salesforce Extensions.
teiid_mongo	MongoDB Extensions
teiid_odata	OData Extensions
teiid_accumulo	Accumulo Extensions
teiid_excel	Excel Extensions

Prefix	Description
teiid_ldap	LDAP Extensions
teiid_rest	REST Extensions
teiid_pi	PI Database Extensions

2.3. DDL METADATA FOR DOMAINS

Domains are simple type declarations that define a set of valid values for a given type name. They can be created at the database level only.

Create domain

```
CREATE DOMAIN <Domain name> [ AS ] <data type>
    [ [NOT] NULL ]
```

The domain name may any non-keyword identifier.

See the BNF for Data Types

Once a domain is defined it may be referenced as the data type for a column, parameter, etc.

Example: Virtual database DDL

```
CREATE DOMAIN mychar AS VARCHAR(1000);

CREATE VIRTUAL SCHEMA viewLayer;
SET SCHEMA viewLayer;
CREATE VIEW v1 (col1 mychar) as select 'value';
...
```

When the system metadata is queried, the type for the column is shown as the domain name.

Limitations

Domain names might not be recognized in the following places where a data type is expected:

- create temp table
- execute immediate
- arraytable
- objecttable
- texttable
- xmltable

When you query a pg_attribute, the ODBC/pg metadata will show the name of the base type, rather than the domain name.

CHAPTER 3. SQL COMPATIBILITY

Data Virtualization provides nearly all of the functionality of SQL-92 DML. SQL-99 and later features are constantly being added based upon community need. The following does not attempt to cover SQL exhaustively, but rather highlights how SQL is used within Data Virtualization. For details about the exact form of SQL that Data Virtualization accepts, see the BNF for SQL grammar.

3.1. IDENTIFIERS

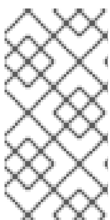
SQL commands contain references to tables and columns. These references are in the form of identifiers, which uniquely identify the tables and columns in the context of the command. All queries are processed in the context of a virtual database, or VDB. Because information can be federated across multiple sources, tables and columns must be scoped in some manner to avoid conflicts. This scoping is provided by schemas, which contain the information for each data source or set of views.

Fully-qualified table and column names are of the following form, where the separate `parts' of the identifier are delimited by periods.

- TABLE: <schema_name>.<table_spec>
- COLUMN: <schema_name>.<table_spec>.<column_name>

Syntax rules

- Identifiers can consist of alphanumeric characters, or the underscore (_) character, and must begin with an alphabetic character. Any Unicode character may be used in an identifier.
- Identifiers in double quotes can have any contents. The double quote character can be used if is escaped with an additional double quote; for example, **"some "" id"**
- Because different data sources organize tables in different ways, with some prepending catalog, schema, or user information, Data Virtualization allows table specification to be a dot-delimited construct.



NOTE

When a table specification contains a dot resolving will allow for the match of a partial name against any number of the end segments in the name. e.g. a table with the fully-qualified name **vdbname."sourceschema.sourcetable"** would match the partial name **sourcetable**.

- Columns, column aliases, and schemas cannot contain a dot (**.**) character.
- Identifiers, even when quoted, are not case-sensitive in Data Virtualization.

Some examples of valid, fully-qualified table identifiers are:

- MySchema.Portfolios
- "MySchema.Portfolios"
- MySchema.MyCatalog.dbo.Authors

Some examples of valid fully-qualified column identifiers are:

- MySchema.Portfolios.portfolioID
- "MySchema.Portfolios"."portfolioID"
- MySchema.MyCatalog.dbo.Authors.lastName

Fully-qualified identifiers can always be used in SQL commands. Partially- or unqualified forms can also be used, as long as the resulting names are unambiguous in the context of the command. Different forms of qualification can be mixed in the same query.

If you use an alias containing a period (**.**) character, it is a known issue that the alias name will be treated the same as a qualified name and may conflict with fully qualified object names.

Reserved words

Reserved words in Data Virtualization include the standard SQL 2003 Foundation, SQL/MED, and SQL/XML reserved words, as well as Data Virtualization specific words such as BIGINTEGER, BIGDECIMAL, or MAKEDEP. For more information about reserved words, see the *Reserved Keywords* and *Reserved Keywords For Future Use* sections in BNF for SQL grammar.

3.2. OPERATOR PRECEDENCE

Data Virtualization parses and evaluates operators with higher precedence before those with lower precedence. Operators with equal precedence are left-associative (left-to-right). The following table lists operator precedence from high to low:

Operator	Description
[]	array element reference
+,-	positive/negative value expression
*,/	multiplication/division
+,-	addition/subtraction
||	concat
criteria	For information, see Criteria.

3.3. EXPRESSIONS

Identifiers, literals, and functions can be combined into expressions. Expressions can be used in a query with nearly any keyword, including SELECT, FROM (if specifying join criteria), WHERE, GROUP BY, HAVING, or ORDER BY.

You can use following types of expressions in Data Virtualization:

- Column identifiers
- Literals

- Aggregate functions
- Window functions
- Case and searched case
- Scalar subqueries
- Parameter references
- Arrays
- Criteria
- Scalar functions

3.3.1. Column Identifiers

Column identifiers are used to specify the output columns in SELECT statements, the columns and their values for INSERT and UPDATE statements, and criteria used in WHERE and FROM clauses. They are also used in GROUP BY, HAVING, and ORDER BY clauses. The syntax for column identifiers was defined in the Identifiers section above.

3.3.2. Literals

Literal values represent fixed values. These can be any of the 'standard' data types. For information about data types, see Data types.

Syntax rules

- Integer values will be assigned an integral data type big enough to hold the value (integer, long, or biginteger).
- Floating point values will always be parsed as a double.
- The keyword 'null' is used to represent an absent or unknown value and is inherently untyped. In many cases, a null literal value will be assigned an implied type based on context. For example, in the function '5 + null', the null value will be assigned the type 'integer' to match the type of the value '5'. A null literal used in the SELECT clause of a query with no implied context will be assigned to type 'string'.

Some examples of simple literal values are:

```
'abc'
```

Example: Escaped single tick

```
'isn''t true'
```

```
5
```

Example: Scientific notation

```
-37.75e01
```

-

Example: exact numeric type BigDecimal

```
100.0
```

```
true
```

```
false
```

Example: Unicode character

```
'\u0027'
```

Example: Binary

```
X'0F0A'
```

Date/Time literals can use either JDBC Escaped literal syntax:

Example: Date literal

```
{d'...'}
```

Example: Time literal

```
{t'...'}
```

Example: Timestamp literal

```
{ts'...'}
```

Or the ANSI keyword syntax:

Example: Date literal

```
DATE '...'
```

Example: Time literal

```
TIME '...'
```

Example: Timestamp literal

```
TIMESTAMP '...'
```

Either way, the string literal value portion of the expression is expected to follow the defined format - "yyyy-MM-dd" for date, "hh:mm:ss" for time, and "yyyy-MM-dd[hh:mm:ss[.fff...]]" for timestamp.

Aggregate functions

Aggregate functions take sets of values from a group produced by an explicit or implicit GROUP BY and return a single scalar value computed from the group.

You can use the following aggregate functions in Data Virtualization:

COUNT(*)
: Count the number of values (including nulls and duplicates) in a group. Returns an integer - an exception will be thrown if a larger count is computed.

COUNT(x)
: Count the number of values (excluding nulls) in a group. Returns an integer - an exception will be thrown if a larger count is computed.

COUNT_BIG(*)
: Count the number of values (including nulls and duplicates) in a group. Returns a long - an exception will be thrown if a larger count is computed.

COUNT_BIG(x)
: Count the number of values (excluding nulls) in a group. Returns a long - an exception will be thrown if a larger count is computed.

SUM(x)
: Sum of the values (excluding nulls) in a group.

AVG(x)
: Average of the values (excluding nulls) in a group.

MIN(x)
: Minimum value in a group (excluding null).

MAX(x)
: Maximum value in a group (excluding null).

ANY(x)/SOME(x)
: Returns TRUE if any value in the group is TRUE (excluding null).

EVERY(x)
: Returns TRUE if every value in the group is TRUE (excluding null).

VAR_POP(x)
: Biased variance (excluding null) logically equals(sum(x^2) - sum(x)^2/count(x))/count(x); returns a double; null if count = 0.

VAR_SAMP(x)
: Sample variance (excluding null) logically equals(sum(x^2) - sum(x)^2/count(x))/(count(x) - 1); returns a double; null if count < 2.

STDDEV_POP(x)
: Standard deviation (excluding null) logically equals SQRT(VAR_POP(x)).

STDDEV_SAMP(x)
: Sample standard deviation (excluding null) logically equals SQRT(VAR_SAMP(x)).

TEXTAGG(expression [as name], ... [DELIMITER char] [QUOTE char | NO QUOTE] [HEADER] [ENCODING id] [ORDER BY ...])
: CSV text aggregation of all expressions in each row of a group. When DELIMITER is not specified, by default comma(,) is used as delimiter. All non-null values will be quoted. Double quotes(") is the default quote character. Use QUOTE to specify a different value, or NO QUOTE for no value quoting. If HEADER is specified, the result contains the header row as the first line - the header line will be present even if there are no rows in a group. This aggregation returns a blob.

```
TEXTAGG(col1, col2 as name DELIMITER '|' HEADER ORDER BY col1)
```

- XMLAGG(xml_expr [ORDER BY ...]) – XML concatenation of all XML expressions in a group (excluding null). The ORDER BY clause cannot reference alias names or use positional ordering.
- JSONARRAY_AGG(x [ORDER BY ...]) – creates a JSON array result as a Clob including null value. The ORDER BY clause cannot reference alias names or use positional ordering. For more information, see JSONARRAY function.

Example: Integer value expression

```
jsonArray_Agg(col1 order by col1 nulls first)
```

could return

```
[null,null,1,2,3]
```

- STRING_AGG(x, delim) – creates a lob results from the concatenation of x using the delimiter delim. If either argument is null, no value is concatenated. Both arguments are expected to be character (string/clob) or binary (varbinary, blob), and the result will be CLOB or BLOB respectively. DISTINCT and ORDER BY are allowed in STRING_AGG.

Example: String aggregate expression

```
string_agg(col1, ',' ORDER BY col1 ASC)
```

could return

```
'a,b,c'
```

- LIST_AGG(x [, delim]) WITHIN GROUP (ORDER BY ...) – a form of STRING_AGG that uses the same syntax as Oracle. Here **x** can be any type that can be converted to a string. The **delim** value, if specified, must be a literal, and the **ORDER BY** value is required. This is only a parsing alias for an equivalent **string_agg** expression.

Example: List aggregate expression

```
listagg(col1, ',') WITHIN GROUP (ORDER BY col1 ASC)
```

could return

```
'a,b,c'
```

- ARRAY_AGG(x [ORDER BY ...]) – Creates an array with a base type that matches the expression x. The ORDER BY clause cannot reference alias names or use positional ordering.
- agg([DISTINCT|ALL] arg ... [ORDER BY ...]) – A user defined aggregate function.

Syntax rules

- Some aggregate functions may contain a keyword 'DISTINCT' before the expression, indicating that duplicate expression values should be ignored. DISTINCT is not allowed in COUNT(*) and is not meaningful in MIN or MAX (result would be unchanged), so it can be used in COUNT, SUM,

and AVG.

- Aggregate functions cannot be used in FROM, GROUP BY, or WHERE clauses without an intervening query expression.
- Aggregate functions cannot be nested within another aggregate function without an intervening query expression.
- Aggregate functions may be nested inside other functions.
- Any aggregate function may take an optional FILTER clause of the form

```
FILTER ( WHERE condition )
```

The condition may be any boolean value expression that does not contain a subquery or a correlated variable. The filter will logically be evaluated for each row prior to the grouping operation. If false the aggregate function will not accumulate a value for the given row.

For more information on aggregates, see the sections on GROUP BY or HAVING.

3.3.3. Window functions

Data Virtualization provides ANSI SQL 2003 window functions. A window function allows an aggregate function to be applied to a subset of the result set, without the need for a **GROUP BY** clause. A window function is similar to an aggregate function, but requires the use of an **OVER** clause or window specification.

Usage:

```
  aggregate [FILTER (WHERE ...)] OVER ( [partition] [ORDER BY ...] [frame] )
| FIRST_VALUE(val) OVER ( [partition] [ORDER BY ...] [frame] )
| LAST_VALUE(val) OVER ( [partition] [ORDER BY ...] [frame] )
| analytical OVER ( [partition] [ORDER BY ...] )

partition := PARTITION BY expression [, expression]*

frame := range_or_rows extent

range_or_rows := RANGE | ROWS

extent :=
    frameBound
  | BETWEEN frameBound AND frameBound

frameBound :=
    UNBOUNDED PRECEDING
  | UNBOUNDED FOLLOWING
  | n PRECEDING
  | n FOLLOWING
  | CURRENT ROW
```

In the preceding syntax, **aggregate** can refer to any aggregate function. Keywords exist for the following analytical functions ROW_NUMBER, RANK, DENSE_RANK, PERCENT_RANK, CUME_DIST. There are also the FIRST_VALUE, LAST_VALUE, LEAD, LAG, NTH_VALUE, and NTILE analytical functions. For more information, see Analytical functions definitions.

Syntax rules

- Window functions can only appear in the SELECT and ORDER BY clauses of a query expression.
- Window functions cannot be nested in one another.
- Partitioning and order by expressions cannot contain subqueries or outer references.
- An aggregate ORDER BY clause cannot be used when windowed.
- The window specification ORDER BY clause cannot reference alias names or use positional ordering.
- Windowed aggregates may not use DISTINCT if the window specification is ordered.
- Analytical value functions may not use DISTINCT and require the use of an ordering in the window specification.
- RANGE or ROWS requires the ORDER BY clause to be specified. The default frame if not specified is RANGE UNBOUNDED PRECEDING. If no end is specified the default is CURRENT ROW. No combination of start and end is allowed such that the end is before the start - for example UNBOUNDED FOLLOWING is not allow as a start nor is UNBOUNDED PRECEDING allowed as an end.
- RANGE cannot be used n PRECEDING or n FOLLOWING

Analytical function definitions

Ranking functions

- RANK() – Assigns a number to each unique ordering value within each partition starting at 1, such that the next rank is equal to the count of prior rows.
- DENSE_RANK() – Assigns a number to each unique ordering value within each partition starting at 1, such that the next rank is sequential.
- PERCENT_RANK() – Computed as (RANK - 1) / (RC - 1) where RC is the total row count of the partition.
- CUME_DIST() – Computed as the PR / RC where PR is the rank of the row including peers and RC is the total row count of the partition.
  By default all values are integers - an exception will be thrown if a larger value is needed. Use the system org.teiid.longRanks to have RANK, DENSE_RANK, and ROW_NUMBER return long values instead.

Value functions

- FIRST_VALUE(val) – Return the first value in the window frame with the given ordering.
- LAST_VALUE(val) – Return the last observed value in the window frame with the given ordering.
- LEAD(val [, offset [, default]]) - Access the ordered value in the window that is offset rows ahead of the current row. If there is no such row, then the default value will be returned. If not specified the offset is 1 and the default is null.

- LAG(val [, offset [, default]]) - Access the ordered value in the window that is offset rows behind of the current row. If there is no such row, then the default value will be returned. If not specified the offset is 1 and the default is null.

- NTH_VALUE(val, n) - Returns the nth val in window frame. The index must be greater than 0. If no such value exists, then null is returned.

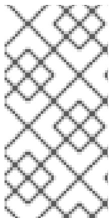
Row value functions

- ROW_NUMBER() – Sequentially assigns a number to each row in a partition starting at **1**.

- NTILE(n) – Divides the partition into n tiles that differ in size by at most **1**. Larger tiles will be created sequentially starting at the first. **n** must be greater than **0**.

Processing

Window functions are logically processed just before creating the output from the SELECT clause. Window functions can use nested aggregates if a GROUP BY clause is present. There is no guaranteed effect on the output ordering from the presence of window functions. The SELECT statement must have an ORDER BY clause to have a predictable ordering.



NOTE

An ORDER BY in the OVER clause follows the same rules pushdown and processing rules as a top level ORDER BY. In general this means you should specify NULLS FIRST/LAST as null handling may differ between engine and pushdown processing. Also see the system properties controlling sort behavior if you different default behavior.

Data Virtualization processes all window functions with the same window specification together. In general, a full pass over the row values coming into the SELECT clause is required for each unique window specification. For each window specification the values are grouped according to the PARTITION BY clause. If no PARTITION BY clause is specified, then the entire input is treated as a single partition.

The frame for the output value is determined based upon the definition of the analytical function or the **ROWS/RANGE** clause. The default frame is **RANGE UNBOUNDED PRECEDING**, which also implies the default end bound of **CURRENT ROW**. **RANGE** computes over a row and its peers together. **ROWS** computes over every row. Most analytical functions, such as **ROW_NUMBER**, have an implicit **RANGE/ROWS** - which is why a different one cannot be specified. For example, **ROW_NUMBER() OVER (order)`** can be expressed instead as **count(*) OVER (order ROWS UNBOUNDED PRECEDING AND CURRENT ROW)**. Thus it assigns a different value to every row regardless of the number of peers.

Example: Windowed results

```
SELECT name, salary, max(salary) over (partition by name) as max_sal,
       rank() over (order by salary) as rank, dense_rank() over (order by salary) as dense_rank,
       row_number() over (order by salary) as row_num FROM employees
```

name	salary	max_sal	rank	dense_rank	row_num
John	100000	100000	2	2	2
Henry	50000	50000	5	4	5

name	salary	max_sal	rank	dense_rank	row_num
John	60000	100000	3	3	3
Suzie	60000	150000	3	3	4
Suzie	150000	150000	1	1	1

3.3.4. Case and searched case

In Data Virtualization, to include conditional logic in a scalar expression, you can use the following two forms of the CASE expression:

- **CASE <expr> (WHEN <expr> THEN <expr>)+ [ELSE expr] END**
- **CASE (WHEN <criteria> THEN <expr>)+ [ELSE expr] END**

Each form allows for an output based on conditional logic. The first form starts with an initial expression and evaluates WHEN expressions until the values match, and outputs the THEN expression. If no WHEN is matched, the ELSE expression is output. If no WHEN is matched and no ELSE is specified, a null literal value is output. The second form (the searched case expression) searches the WHEN clauses, which specify an arbitrary criteria to evaluate. If any criteria evaluates to true, the THEN expression is evaluated and output. If no WHEN is true, the ELSE is evaluated or NULL is output if none exists.

Example case statements

```
SELECT CASE columnA WHEN '10' THEN 'ten' WHEN '20' THEN 'twenty' END AS myExample

SELECT CASE WHEN columnA = '10' THEN 'ten' WHEN columnA = '20' THEN 'twenty' END AS
myExample
```

3.3.5. Scalar subqueries

Subqueries can be used to produce a single scalar value in the SELECT, WHERE, or HAVING clauses only. A scalar subquery must have a single column in the SELECT clause and should return either 0 or 1 row. If no rows are returned, null will be returned as the scalar subquery value. For information about other types of subqueries, see Subqueries.

3.3.6. Parameter references

Parameters are specified using a **?** symbol. You can use parameters only with **PreparedStatement** or **CallableStatements** in JDBC. Each parameter is linked to a value specified by 1-based index in the JDBC API.

3.3.7. Arrays

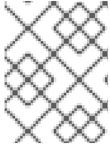
Array values may be constructed using parentheses around an expression list with an optional trailing comma, or with an explicit ARRAY constructor.

Example: Empty arrays

```
()
(,)
ARRAY[]
```

Example: Single element array

```
(expr,)
ARRAY[expr]
```



NOTE

A trailing comma is required for the parser to recognize a single element expression as an array with parentheses, rather than a simple nested expression.

Example: General array syntax

```
(expr, expr ... [,])
ARRAY[expr, ...]
```

If all of the elements in the array have the same type, the array will have a matching base type. If the element types differ the array base type will be object.

An array element reference takes the form of:

```
array_expr[index_expr]
```

index_expr must resolve to an integer value. This syntax is effectively the same as the **array_get** system function and expects 1-based indexing.

3.4. CRITERIA

Criteria can be any of the following items:

- Predicates that evaluate to true or false.
- Logical criteria that combine criteria (AND, OR, NOT).
- A value expression of type Boolean.

Usage

```
criteria AND|OR criteria
```

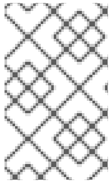
```
NOT criteria
```

```
(criteria)
```

```
expression (=|<>|!=|<|>|<=|>=) (expression|((ANY|ALL|SOME) subquery|(array_expression)))
```

```
expression IS [NOT] DISTINCT FROM expression
```

IS DISTINCT FROM considers null values to be equivalent and never produces an UNKNOWN value.



NOTE

Because the optimizer is not tuned to handle **IS DISTINCT FROM**, if you use it in a join predicate that is not pushed down, the resulting plan does not perform as well a regular comparison.

```
expression [NOT] IS NULL
```

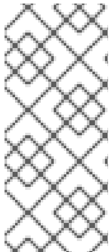
```
expression [NOT] IN (expression [,expression]*)|subquery
```

```
expression [NOT] LIKE pattern [ESCAPE char]
```

LIKE matches the string expression against the given string pattern. The pattern may contain **%** to match any number of characters, and _ to match any single character. The escape character can be used to escape the match characters **%** and _.

```
expression [NOT] SIMILAR TO pattern [ESCAPE char]
```

SIMILAR TO is a cross between LIKE and standard regular expression syntax. **%** and _ are still used, rather than **.*** and **.**, respectively.

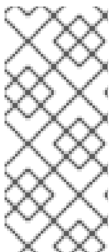


NOTE

Data Virtualization does not exhaustively validate **SIMILAR TO** pattern values. Instead, the pattern is converted to an equivalent regular expression. Do not rely on general regular expression features when using **SIMILAR TO**. If additional features are needed, use **LIKE_REGEX**. Avoid the use of non-literal patterns, because Data Virtualization has a limited ability to process SQL pushdown predicates.

```
expression [NOT] LIKE_REGEX pattern
```

You can use **LIKE_REGEX** with standard regular expression syntax for matching. This differs from **SIMILAR TO** and **LIKE** in that the escape character is no longer used. \ is already the standard escape mechanism in regular expressions, and **%**` and _ have no special meaning. The runtime engine uses the JRE implementation of regular expressions. For more information, see the java.util.regex.Pattern class.



NOTE

Data Virtualization does not exhaustively validate **LIKE_REGEX** pattern values. It is possible to use JRE-only regular expression features that are not specified by the SQL specification. Additionally, not all sources can use the same regular expression flavor or extensions. In pushdown situations, be careful to ensure that the pattern that you use has the same meaning in Data Virtualization, and across all applicable sources.

```
EXISTS (subquery)
```

```
expression [NOT] BETWEEN minExpression AND maxExpression
```

Data Virtualization converts **BETWEEN** into the equivalent form **expression >= minExpression AND expression ⇐ maxExpression**.

> expression

Where **expression** has type Boolean.

Syntax rules

- The precedence ordering from lowest to highest is comparison, NOT, AND, OR.
- Criteria nested by parenthesis will be logically evaluated prior to evaluating the parent criteria.

Some examples of valid criteria are:

- **(balance > 2500.0)**
- **100*(50 - x)/(25 - y) > z**
- **concat(areaCode,concat('-',phone)) LIKE '314%1'**

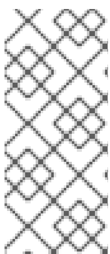
COMPARING NULL VALUES

Null values represent an unknown value. Comparison with a null value will evaluate to **unknown**, which can never be true even if **not** is used.

Criteria precedence

Data Virtualization parses and evaluates conditions with higher precedence before those with lower precedence. Conditions with equal precedence are left-associative. The following table lists condition precedence from high to low:

Condition	Description
SQL operators	See Expressions
EXISTS, LIKE, SIMILAR TO, LIKE_REGEX, BETWEEN, IN, IS NULL, IS DISTINCT, <, ⇐, >, >=, =, <>	Comparison
NOT	Negation
AND	Conjunction
OR	Disjunction



NOTE

To prevent lookaheads, the parser does not accept all possible criteria sequences. For example, **a = b is null** is not accepted, because by the left-associative parsing we first recognize **a =**, then look for a common value expression. **b is null** is not a valid common value expression. Thus, nesting must be used, for example, **(a = b) is null**. For more information about parsing rules, see BNF for SQL grammar.

3.5. SCALAR FUNCTIONS

Data Virtualization provides an extensive set of built-in scalar functions. For more information, see DML commands and Data types. In addition, Data Virtualization provides the capability for user-defined functions or UDFs. For information about adding UDFs, see *User-defined functions* in the Translator Development Guide. After you add UDFs, you can call them in the same way that you call other functions.

3.5.1. Numeric functions

Numeric functions return numeric values (integer, long, float, double, biginteger, bigdecimal). They generally take numeric values as inputs, though some take strings.

Function	Definition	Datatype constraint
+ - * /	Standard numeric operators	x in {integer, long, float, double, biginteger, bigdecimal}, return type is same as x [a]
ABS(x)	Absolute value of x	See standard numeric operators above
ACOS(x)	Arc cosine of x	x in {double, bigdecimal}, return type is double
ASIN(x)	Arc sine of x	x in {double, bigdecimal}, return type is double
ATAN(x)	Arc tangent of x	x in {double, bigdecimal}, return type is double
ATAN2(x,y)	Arc tangent of x and y	x, y in {double, bigdecimal}, return type is double
CEILING(x)	Ceiling of x	x in {double, float}, return type is double
COS(x)	Cosine of x	x in {double, bigdecimal}, return type is double
COT(x)	Cotangent of x	x in {double, bigdecimal}, return type is double
DEGREES(x)	Convert x degrees to radians	x in {double, bigdecimal}, return type is double
EXP(x)	e^x	x in {double, float}, return type is double

Function	Definition	Datatype constraint
FLOOR(x)	Floor of x	x in {double, float}, return type is double
FORMATBIGDECIMAL(x, y)	Formats x using format y	x is bigdecimal, y is string, returns string
FORMATBIGINTEGER(x, y)	Formats x using format y	x is biginteger, y is string, returns string
FORMATDOUBLE(x, y)	Formats x using format y	x is double, y is string, returns string
FORMATFLOAT(x, y)	Formats x using format y	x is float, y is string, returns string
FORMATINTEGER(x, y)	Formats x using format y	x is integer, y is string, returns string
FORMATLONG(x, y)	Formats x using format y	x is long, y is string, returns string
LOG(x)	Natural log of x (base e)	x in {double, float}, return type is double
LOG10(x)	Log of x (base 10)	x in {double, float}, return type is double
MOD(x, y)	Modulus (remainder of x / y)	x in {integer, long, float, double, biginteger, bigdecimal}, return type is same as x
PARSEBIGDECIMAL(x, y)	Parses x using format y	x, y are strings, returns bigdecimal
PARSEBIGINTEGER(x, y)	Parses x using format y	x, y are strings, returns biginteger
PARSEDOUBLE(x, y)	Parses x using format y	x, y are strings, returns double
PARSEFLOAT(x, y)	Parses x using format y	x, y are strings, returns float
PARSEINTEGER(x, y)	Parses x using format y	x, y are strings, returns integer
PARSELONG(x, y)	Parses x using format y	x, y are strings, returns long
PI()	Value of Pi	return is double
POWER(x,y)	x to the y power	x in {double, bigdecimal, biginteger}, return is the same type as x

Function	Definition	Datatype constraint
RADIANS(x)	Convert x radians to degrees	x in {double, bigdecimal}, return type is double
RAND()	Returns a random number, using generator established so far in the query or initializing with system clock if necessary.	Returns double.
RAND(x)	Returns a random number, using new generator seeded with x. This should typically be called in an initialization query. It will only effect the random values returned by the Data Virtualization RAND function and not the values from RAND functions evaluated by sources.	x is integer, returns double.
ROUND(x,y)	Round x to y places; negative values of y indicate places to the left of the decimal point	x in {integer, float, double, bigdecimal} y is integer, return is same type as x.
SIGN(x)	1 if x > 0, 0 if x = 0, -1 if x < 0	x in {integer, long, float, double, biginteger, bigdecimal}, return type is integer
SIN(x)	Sine value of x	x in {double, bigdecimal}, return type is double
SQRT(x)	Square root of x	x in {long, double, bigdecimal}, return type is double
TAN(x)	Tangent of x	x in {double, bigdecimal}, return type is double
BITAND(x, y)	Bitwise AND of x and y	x, y in {integer}, return type is integer
BITOR(x, y)	Bitwise OR of x and y	x, y in {integer}, return type is integer
BITXOR(x, y)	Bitwise XOR of x and y	x, y in {integer}, return type is integer
BITNOT(x)	Bitwise NOT of x	x in {integer}, return type is integer

[a] The precision and scale of non-bigdecimal arithmetic function functions results matches that of Java. The results of bigdecimal operations match Java, except for division, which uses a preferred scale of max(16, dividend.scale + divisor.precision + 1), which then has trailing zeros removed by setting the scale to max(dividend.scale, normalized scale).

Parsing numeric datatypes from strings

Data Virtualization offers a set of functions you can use to parse numbers from strings. For each string, you need to provide the formatting of the string. These functions use the convention established by the java.text.DecimalFormat class to define the formats you can use with these functions. You can learn more about how this class defines numeric string formats by visiting the Sun Java Web site at the following URL for Sun Java .

For example, you could use these function calls, with the formatting string that adheres to the java.text.DecimalFormat convention, to parse strings and return the datatype you need:

Input String	Function Call to Format String	Output Value	Output Datatype
'$25.30'	parseDouble(cost, '$,0.00;($,0.00)')	25.3	double
'25%'	parseFloat(percent, ',#0%')	25	float
'2,534.1'	parseFloat(total, ',0.;-,0.')	2534.1	float
'1.234E3'	parseLong(amt, '0.###E0')	1234	long
'1,234,567'	parseInteger(total, ',0;-,0')	1234567	integer

Formatting numeric datatypes as strings

Data Virtualization offers a set of functions you can use to convert numeric datatypes into strings. For each string, you need to provide the formatting. These functions use the convention established within the java.text.DecimalFormat class to define the formats you can use with these functions. You can learn more about how this class defines numeric string formats by visiting the Sun Java Web site at the following URL for Sun Java .

For example, you could use these function calls, with the formatting string that adheres to the java.text.DecimalFormat convention, to format the numeric datatypes into strings:

Input Value	Input Datatype	Function Call to Format String	Output String
25.3	double	formatDouble(cost, '$,0.00;($,0.00)')	'$25.30'

Input Value	Input Datatype	Function Call to Format String	Output String
25	float	formatFloat(percent, ',#0%')	'25%'
2534.1	float	formatFloat(total, ',0.;-,0.')	'2,534.1'
1234	long	formatLong(amt, '0.###E0')	'1.234E3'
1234567	integer	formatInteger(total, ',0;-,0')	'1,234,567'

3.5.2. String functions

String functions generally take strings as inputs and return strings as outputs.

Unless specified, all of the arguments and return types in the following table are strings and all indexes are 1-based. The 0 index is considered to be before the start of the string.

Function	Definition	Datatype constraint
x || y	Concatenation operator	x,y in {string, clob}, return type is string or character large object (CLOB).
ASCII(x)	Provide ASCII value of the left most character[1] in x. The empty string will as input will return **null**.	return type is integer
CHR(x) CHAR(x)	Provide the character[1] for ASCII value **x** [a].	x in {integer} [1] For the engine's implementations of the ASCII and CHR functions, characters are limited to UCS2 values only. For pushdown there is little consistency among sources for character values beyond character code 255.
CONCAT(x, y)	Concatenates x and y with ANSI semantics. If x and/or y is null, returns null.	x, y in {string}

Function	Definition	Datatype constraint
CONCAT2(x, y)	Concatenates x and y with non-ANSI null semantics. If x and y is null, returns null. If only x or y is null, returns the other value.	x, y in {string}
ENDSWITH(x, y)	Checks if y ends with x. If x or y is null, returns null.	x, y in {string}, returns boolean
INITCAP(x)	Make first letter of each word in string x capital and all others lowercase.	x in {string}
INSERT(str1, start, length, str2)	Insert string2 into string1	str1 in {string}, start in {integer}, length in {integer}, str2 in {string}
LCASE(x)	Lowercase of x	x in {string}
LEFT(x, y)	Get left y characters of x	x in {string}, y in {integer}, return string
LENGTH(x) CHAR_LENGTH(x) CHARACTER_LENGTH(x)	Length of x	return type is integer
LOCATE(x, y) POSITION(x IN y)	Find position of x in y starting at beginning of y.	x in {string}, y in {string}, return integer
LOCATE(x, y, z)	Find position of x in y starting at z.	x in {string}, y in {string}, z in {integer}, return integer
LPAD(x, y)	Pad input string x with spaces on the left to the length of y.	x in {string}, y in {integer}, return string
LPAD(x, y, z)	Pad input string x on the left to the length of y using character z.	x in {string}, y in {string}, z in {character}, return string
LTRIM(x)	Left trim x of blank chars.	x in {string}, return string

Function	Definition	Datatype constraint
QUERYSTRING(path [, expr [AS name] ...])	Returns a properly encoded query string appended to the given path. Null valued expressions are omitted, nd a null path is treated as ''. Names are optional for column reference expressions. For example, **QUERYSTRING('path', 'value' as "&x", ' & ' as y, null as z) returns 'path?%26x=value&y=%20%26%20'**	path, expr in {string}. name is an identifier.
REPEAT(str1,instances)	Repeat string1 a specified number of times	str1 in {string}, instances in {integer} return string.
RIGHT(x, y)	Get right y characters of x	x in {string}, y in {integer}, return string
RPAD(input string x, pad length y)	Pad input string x with spaces on the right to the length of y	x in {string}, y in {integer}, return string
RPAD(x, y, z)	Pad input string x on the right to the length of y using character z	x in {string}, y in {string}, z in {character}, return string
RTRIM(x)	Right trim x of blank chars	x is string, return string
SPACE(x)	Repeat the space character x number of times	x is integer, return string
SUBSTRING(x, y) SUBSTRING(x FROM y)	[b] Get substring from x, from position y to the end of x	y in {integer}
SUBSTRING(x, y, z) SUBSTRING(x FROM y FOR z)	[b] Get substring from x from position y with length z	y, z in {integer}
TRANSLATE(x, y, z)	Translate string x by replacing each character in y with the character in z at the same position.	x in {string}
TRIM([[LEADING|TRAILING|BOTH] [x] FROM] y)	Trim the leading, trailing, or both ends of a string y of character x. If LEADING/TRAILING/BOTH is not specified, BOTH is used. If no trim character x is specified, then the blank space ' is used.	x in {character}, y in {string}
UCASE(x)	Uppercase of x	x in {string}

Function	Definition	Datatype constraint
UNESCAPE(x)	Unescaped version of x. Possible escape sequences are \b - backspace, \t - tab, \n - line feed, \f - form feed, \r - carriage return. \uXXXX, where X is a hex value, can be used to specify any unicode character. \XXX, where X is an octal digit, can be used to specify an octal byte value. If any other character appears after an escape character, that character will appear in the output and the escape character will be ignored.	x in {string}

[a] Non-ASCII range characters or integers used in these functions may produce different results or exceptions depending on where the function is evaluated (Data Virtualization vs. source). Data Virtualization's uses Java default int to char and char to int conversions, which operates over UTF16 values.

[b] The substring function depending upon the source does not have consistent behavior with respect to negative from/length arguments nor out of bounds from/length arguments. The default Data Virtualization behavior is:

- Return a null value when the from value is out of bounds or the length is less than 0
- A zero from index is effective the same as 1.
- A negative from index is first counted from the end of the string.

Some sources, however, can return an empty string instead of **null**, and some sources are not compatible with negative indexing.

TO_CHARS

Return a CLOB from the binary large object (BLOB) with the given encoding.

```
TO_CHARS(x, encoding [, wellformed])
```

BASE64, HEX, UTF-8-BOM and the built-in Java Charset names are valid values for the encoding [b]. x is a BLOB, encoding is a string, wellformed is a boolean, and returns a CLOB. The two argument form defaults to wellformed=true. If wellformed is false, the conversion function will immediately validate the result such that an unmappable character or malformed input will raise an exception.

TO_BYTES

Return a BLOB from the CLOB with the given encoding.

```
TO_BYTES(x, encoding [, wellformed])
```

BASE64, HEX, UTF-8-BOM and the builtin Java Charset names are valid values for the encoding [b]. x in a CLOB, encoding is a string, wellformed is a boolean and returns a BLOB. The two argument form defaults to wellformed=true. If wellformed is false, the conversion function will immediately validate the result such that an unmappable character or malformed input will raise an exception. If wellformed is

true, then unmappable characters will be replaced by the default replacement character for the character set. Binary formats, such as BASE64 and HEX, will be checked for correctness regardless of the wellformed parameter.

[b] For more information about Charset names, see the Charset docs.

REPLACE

Replace all occurrences of a given string with another.

```
REPLACE(x, y, z)
```

Replace all occurrences of y with z in x. x, y, z are strings and the return value is a string.

REGEXP_REPLACE

Replace one or all occurrences of a given pattern with another string.

```
REGEXP_REPLACE(str, pattern, sub [, flags])
```

Replace one or more occurrences of pattern with sub in str. All arguments are strings and the return value is a string.

The pattern parameter is expected to be a valid Java regular expression

The flags argument can be any concatenation of any of the valid flags with the following meanings:

Flag	Name	Meaning
g	Global	Replace all occurrences, not just the first.
m	Multi-line	Match over multiple lines.
i	Case insensitive	Match without case sensitivity.

Usage:

The following will return "xxbye Wxx" using the global and case insensitive options.

Example regexp_replace

```
regexp_replace('Goodbye World', '[g-o].', 'x', 'gi')
```

3.5.3. Date and time functions

Date and time functions return or operate on dates, times, or timestamps.

Date and time functions use the convention established within the java.text.SimpleDateFormat class to define the formats you can use with these functions. You can learn more about how this class defines formats by visiting the Javadocs for SimpleDateFormat.

Function	Definition	Datatype constraint
CURDATE() CURRENT_DATE[()]	Return current date – will return the same value for all invocations in the user command.	returns date.
CURTIME()	Return current time – will return the same value for all invocations in the user command. See also CURRENT_TIME.	returns time
NOW()	Return current timestamp (date and time with millisecond precision) – will return the same value for all invocations in the user command or procedure instruction. See also CURRENT_TIMESTAMP.	returns timestamp
CURRENT_TIME[(precision)]	Return current time – will return the same value for all invocations in the user command. The Data Virtualization time type does not track fractional seconds, so the precision argument is effectively ignored. Without a precision is the same as CURTIME().	returns time
CURRENT_TIMESTAMP[(precision)]	Return current timestamp (date and time with millisecond precision) – will return the same value for all invocations with the same precision in the user command or procedure instruction. Without a precision is the same as NOW(). Since the current timestamp has only millisecond precision by default setting the precision to greater than 3 will have no effect.	returns timestamp
DAYNAME(x)	Return name of day in the default locale	x in {date, timestamp}, returns string
DAYOFMONTH(x)	Return day of month	x in {date, timestamp}, returns integer
DAYOFWEEK(x)	Return day of week (Sunday=1, Saturday=7)	x in {date, timestamp}, returns integer
DAYOFYEAR(x)	Return day number in year	x in {date, timestamp}, returns integer

<table>
<tr><th>Function</th><th>Definition</th><th>Datatype constraint</th></tr>
<tr><td>EPOCH(x)</td><td>Return seconds since the unix epoch with microsecond precision</td><td>x in {date, timestamp}, returns double</td></tr>
<tr><td>EXTRACT(YEAR|MONTH|DAY|HOUR|MINUTE|SECOND|QUARTER|EPOCH FROM x)</td><td>Return the given field value from the date value x. Produces the same result as the associated YEAR, MONTH, DAYOFMONTH, HOUR, MINUTE, SECOND, QUARTER, EPOCH functions functions. The SQL specification also allows for TIMEZONE_HOUR and TIMEZONE_MINUTE as extraction targets. In Data Virtualization all date values are in the timezone of the server.</td><td>x in {date, time, timestamp}, epoch returns double, the others return integer</td></tr>
<tr><td>FORMATDATE(x, y)</td><td>Format date x using format y.</td><td>x is date, y is string, returns string</td></tr>
<tr><td>FORMATTIME(x, y)</td><td>Format time x using format y.</td><td>x is time, y is string, returns string</td></tr>
<tr><td>FORMATTIMESTAMP(x, y)</td><td>Format timestamp x using format y.</td><td>x is timestamp, y is string, returns string</td></tr>
<tr><td>FROM_MILLIS (millis)</td><td>Return the Timestamp value for the given milliseconds.</td><td>long UTC timestamp in milliseconds</td></tr>
<tr><td>FROM_UNIXTIME (unix_timestamp)</td><td>Return the Unix timestamp as a String value with the default format of yyyy/mm/dd hh:mm:ss.</td><td>long Unix timestamp (in seconds)</td></tr>
<tr><td>HOUR(x)</td><td>Return hour (in military 24-hour format).</td><td>x in {time, timestamp}, returns integer</td></tr>
<tr><td>MINUTE(x)</td><td>Return minute.</td><td>x in {time, timestamp}, returns integer</td></tr>
</table>

Function	Definition	Datatype constraint
MODIFYTIMEZONE (timestamp, startTimeZone, endTimeZone)	Returns a timestamp based upon the incoming timestamp adjusted for the differential between the start and end time zones. If the server is in GMT-6, then modifytimezone({ts '2006-01-10 04:00:00.0'},'GMT-7', 'GMT-8') will return the timestamp {ts '2006-01-10 05:00:00.0'} as read in GMT-6. The value has been adjusted 1 hour ahead to compensate for the difference between GMT-7 and GMT-8.	startTimeZone and endTimeZone are strings, returns a timestamp
MODIFYTIMEZONE (timestamp, endTimeZone)	Return a timestamp in the same manner as modifytimezone(timestamp, startTimeZone, endTimeZone), but will assume that the startTimeZone is the same as the server process.	Timestamp is a timestamp; endTimeZone is a string, returns a timestamp
MONTH(x)	Return month.	x in {date, timestamp}, returns integer
MONTHNAME(x)	Return name of month in the default locale.	x in {date, timestamp}, returns string
PARSEDATE(x, y)	Parse date from x using format y.	x, y in {string}, returns date
PARSETIME(x, y)	Parse time from x using format y.	x, y in {string}, returns time
PARSETIMESTAMP(x,y)	Parse timestamp from x using format y.	x, y in {string}, returns timestamp
QUARTER(x)	Return quarter.	x in {date, timestamp}, returns integer
SECOND(x)	Return seconds.	x in {time, timestamp}, returns integer
TIMESTAMPCREATE(date, time)	Create a timestamp from a date and time.	date in {date}, time in {time}, returns timestamp
TO_MILLIS (timestamp)	Return the UTC timestamp in milliseconds.	timestamp value

Function	Definition	Datatype constraint
UNIX_TIMESTAMP (unix_timestamp)	Return the long Unix timestamp (in seconds).	unix_timestamp String in the default format of yyyy/mm/dd hh:mm:ss
WEEK(x)	Return week in year 1-53. For customization information, see *System Properties* in the Administrator's Guide.	x in {date, timestamp}, returns integer
YEAR(x)	Return four-digit year	x in {date, timestamp}, returns integer

Timestampadd

Add a specified interval amount to the timestamp.

Syntax

```
TIMESTAMPADD(interval, count, timestamp)
```

Arguments

Name	Description
interval	A datetime interval unit, can be one of the following keywords: • SQL_TSI_FRAC_SECOND - fractional seconds (billionths of a second) • SQL_TSI_SECOND - seconds • SQL_TSI_MINUTE - minutes • SQL_TSI_HOUR - hours • SQL_TSI_DAY - days • SQL_TSI_WEEK - weeks using Sunday as the first day • SQL_TSI_MONTH - months • SQL_TSI_QUARTER - quarters (3 months) where the first quarter is months 1-3, etc. • SQL_TSI_YEAR - years
count	A long or integer count of units to add to the timestamp. Negative values will subtract that number of units. Long values are allowed for symmetry with TIMESTAMPDIFF - but the effective range is still limited to integer values.
timestamp	A datetime expression.

Example

```
SELECT TIMESTAMPADD(SQL_TSI_MONTH, 12,'2016-10-10')
SELECT TIMESTAMPADD(SQL_TSI_SECOND, 12,'2016-10-10 23:59:59')
```

Timestampdiff

Calculates the number of date part intervals crossed between the two timestamps return a long value.

Syntax

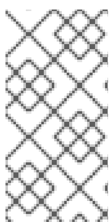
```
TIMESTAMPDIFF(interval, startTime, endTime)
```

Arguments

Name	Description
interval	A datetime interval unit, the same as keywords used by Timestampadd.
startTime	A datetime expression.
endTime	A datetime expression.

Example

```
SELECT TIMESTAMPDIFF(SQL_TSI_MONTH,'2000-01-02','2016-10-10')
SELECT TIMESTAMPDIFF(SQL_TSI_SECOND,'2000-01-02 00:00:00','2016-10-10 23:59:59')
SELECT TIMESTAMPDIFF(SQL_TSI_FRAC_SECOND,'2000-01-02 00:00:00.0','2016-10-10
23:59:59.999999')
```



NOTE

If (endTime > startTime), a non-negative number will be returned. If (endTime < startTime), a non-positive number will be returned. The date part difference difference is counted regardless of how close the timestamps are. For example, '2000-01-02 00:00:00.0' is still considered 1 hour ahead of '2000-01-01 23:59:59.999999'.

Compatibility issues

- In SQL, Timestampdiff typically returns an integer. However the Data Virtualization implementation returns a long. You might receive an exception if you expect a value out of the integer range from a pushed down timestampdiff.
- The implementation of timestamp diff in Teiid 8.2 and earlier versions returned values based on the number of whole canonical interval approximations (365 days in a year, 91 days in a quarter, 30 days in a month, etc.) crossed. For example the difference in months between 2013-03-24 and 2013-04-01 was 0, but based upon the date parts crossed is 1. For information about backwards compatibility, see *System Properties* in the Adminstrator's Guide.

Parsing date datatypes from strings

Data Virtualization does not implicitly convert strings that contain dates presented in different formats, such as '19970101' and '31/1/1996' to date-related datatypes. You can, however, use the parseDate, parseTime, and parseTimestamp functions, described in the next section, to explicitly convert strings with a different format to the appropriate datatype. These functions use the convention established within the java.text.SimpleDateFormat class to define the formats you can use with these functions. For more information about how this class defines date and time string formats, see Javadocs for SimpleDateFormat. Note that the format strings are specific to your Java default locale.

For example, you could use these function calls, with the formatting string that adheres to the java.text.SimpleDateFormat convention, to parse strings and return the datatype you need:

String	Function call to parse string
'1997010'	parseDate(myDateString, 'yyyyMMdd')
'31/1/1996'	parseDate(myDateString, 'dd''/''MM''/''yyyy')
'22:08:56 CST'	parseTime (myTime, 'HH:mm:ss z')
'03.24.2003 at 06:14:32'	parseTimestamp(myTimestamp, 'MM.dd.yyyy''at''hh:mm:ss')

Specifying time zones

Time zones can be specified in several formats. Common abbreviations such as EST for "Eastern standard time" are allowed but discouraged, as they can be ambiguous. Unambiguous time zones are defined in the form continent or ocean/largest city. For example, America/New_York, America/Buenos_Aires, or Europe/London. sAdditionally, you can specify a custom time zone by GMT offset: GMT[+/-]HH:MM.

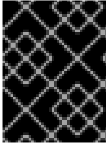
For example: GMT-05:00

3.5.4. Type conversion functions

Within your queries, you can convert between datatypes using the CONVERT or CAST keyword. For more information, see Type conversions

Function	Definition
CONVERT(x, type)	Convert x to type, where type is a Data Virtualization Base Type
CAST(x AS type)	Convert x to type, where type is a Data Virtualization Base Type

These functions are identical other than syntax; CAST is the standard SQL syntax, CONVERT is the standard JDBC/ODBC syntax.



IMPORTANT

Options that are specified on the type, such as length, precision, scale, etc., are effectively ignored - the runtime is simply converting from one object type to another.

3.5.5. Choice functions

Choice functions provide a way to select from two values based on some characteristic of one of the values.

Function	Definition	Datatype constraint
COALESCE(x,y+)	Returns the first non-null parameter.	x and all y's can be any compatible types.
IFNULL(x,y)	If x is null, return y; else return x.	x, y, and the return type must be the same type but can be any type.
NVL(x,y)	If x is null, return y; else return x.	x, y, and the return type must be the same type but can be any type.
NULLIF(param1, param2)	Equivalent to case when (param1 = param2) then null else param1.	param1 and param2 must be compatable comparable types.

IFNULL and NVL are aliases of each other. They are the same function.

3.5.6. Decode functions

Decode functions allow you to have the Data Virtualization server examine the contents of a column in a result set and alter, or decode, the value so that your application can better use the results.

Function	Definition	Datatype constraint
DECODESTRING(x, y [, z])	Decode column **x** using string of value pairs y with optional delimiter **z** and return the decoded column as a string. If a delimiter is not specified, a comma (**,**) is used. **y** has the format **SearchDelimResultDelimSearchDelimResult[DelimDefault]**. Returns Default if specified or**x** if there are no matches. **Deprecated. Use a CASE expression instead.**	all string

Function	Definition	Datatype constraint
DECODEINTEGER(x, y [, z])	Decode column **x** using string of value pairs **y** with optional delimiter z and return the decoded column as an integer. If a delimiter is not specified, a comma(**,**) is used. **y** has the format **SearchDelimResultDelimSearchDelimResult[DelimDefault]**. Returns Default if specified or**x** if there are no matches. **Deprecated. Use a CASE expression instead.**	all string parameters, return integer

Within each function call, you include the following arguments:

1. **x** is the input value for the decode operation. This will generally be a column name.
2. **y** is the literal string that contains a delimited set of input values and output values.
3. **z** is an optional parameter on these methods that allows you to specify what delimiter the string specified in y uses.

For example, your application might query a table called **PARTS** that contains a column called **IS_IN_STOCK**, which contains a Boolean value that you need to change into an integer for your application to process. In this case, you can use the **DECODEINTEGER** function to change the Boolean values to integers:

```
SELECT DECODEINTEGER(IS_IN_STOCK, 'false, 0, true, 1') FROM PartsSupplier.PARTS;
```

When the Data Virtualization system encounters the value **false** in the result set, it replaces the value with 0.

If, instead of using integers, your application requires string values, you can use the **DECODESTRING** function to return the string values you need:

```
SELECT DECODESTRING(IS_IN_STOCK, 'false, no, true, yes, null') FROM PartsSupplier.PARTS;
```

In addition to two input/output value pairs, this sample query provides a value to use if the column does not contain any of the preceding input values. If the row in the IS_IN_STOCK column does not contain true or false, the Data Virtualization server inserts a null into the result set.

When you use these DECODE functions, you can provide as many input/output value pairs if you want within the string. By default, the Data Virtualization system expects a comma delimiter, but you can add a third parameter to the function call to specify a different delimiter:

```
SELECT DECODESTRING(IS_IN_STOCK, 'false:no:true:yes:null',':') FROM PartsSupplier.PARTS;
```

You can use keyword **null** in the DECODE string as either an input value or an output value to represent a null value. However, if you need to use the literal string **null** as an input or output value (which means the word null appears in the column and not a null value) you can put the word in quotes: **"null"**.

```
SELECT DECODESTRING( IS_IN_STOCK, 'null,no,"null",no,nil,no,false,no,true,yes' ) FROM
PartsSupplier.PARTS;
```

If the DECODE function does not find a matching output value in the column and you have not specified a default value, the DECODE function will return the original value the Data Virtualization server found in that column.

3.5.7. Lookup function

The Lookup function provides a way to speed up access to values from a reference table. The Lookup function automatically caches all key and return column pairs declared in the function for the referenced table. Subsequent lookups against the same table using the same key and return columns will use the cached values. This caching accelerates response time to queries that use lookup tables, also known in business terminology as code or reference tables.

```
LOOKUP(codeTable, returnColumn, keyColumn, keyValue)
```

In the lookup table codeTable, find the row where keyColumn has the value keyValue and return the associated returnColumn value or null, if no matching keyValue is found. codeTable must be a string literal that is the fully-qualified name of the target table. returnColumn and keyColumn must also be string literals and match corresponding column names in the codeTable. The keyValue can be any expression that must match the datatype of the keyColumn. The return datatype matches that of returnColumn.

Country code lookup

```
lookup('ISOCountryCodes', 'CountryCode', 'CountryName', 'United States')
```

An ISOCountryCodes table is used to translate a country name to an ISO country code. One column, CountryName, represents the keyColumn. A second column, CountryCode, represents the returnColumn, containing the ISO code of the country. Hence, the usage of the lookup function here will provide a CountryName, shown above as `United States', and expect a CountryCode value in response.

When you call this function for any combination of codeTable, returnColumn, and keyColumn for the first time, the Data Virtualization System caches the result. The Data Virtualization System uses this cache for all queries, in all sessions, that later access this lookup table. You should generally not use the lookup function for data that is subject to updates or may be session/user specific, including row-based security and column masking effects. For more information about caching in the Lookup function, see the Caching Guide .

The keyColumn is expected to contain unique values for its corresponding codeTable. If the keyColumn contains duplicate values, an exception will be thrown.

3.5.8. System functions

System functions provide access to information in the Data Virtualization system from within a query.

COMMANDPAYLOAD

Retrieve a string from the command payload or null if no command payload was specified. The command payload is set by the **TeiidStatement.setPayload** method on the Data Virtualization JDBC API extensions on a per-query basis.

COMMANDPAYLOAD([key])

If the key parameter is provided, the command payload object is cast to a java.util.Properties object, and the corresponding property value for the key is returned. If the key is not specified, the return value is the command payload object toString value.

key, return value are strings

ENV

Retrieve a system property. This function was misnamed and is included for legacy compatibility. See ENV_VAR and SYS_PROP for more appropriately named functions.

ENV(key)

call using ENV('KEY'), which returns value as string. Ex: ENV('PATH'). If a value is not found with the key passed in, a lower cased version of the key is tried as well. This function is treated as deterministic, even though it is possible to set system properties at runtime.

ENV_VAR

Retrieve an environment variable.

ENV_VAR(key)

call using ENV_VAR('KEY'), which returns value as string. Ex: ENV_VAR('USER'). The behavior of this function is platform dependent with respect to case-sensitivity. This function is treated as deterministic, even though it is possible for environment variables to change at runtime.

SYS_PROP

Retrieve an system property.

SYS_PROP(key)

call using SYS_PROP('KEY'), which returns value as string. Ex: SYS_PROP('USER'). This function is treated as deterministic, even though it is possible for system properties to change at runtime.

NODE_ID

Retrieve the node id - typically the System property value for "jboss.node.name" which will not be set for Data Virtualization embedded.

NODE_ID()

return value is string.

SESSION_ID

Retrieve the string form of the current session id.

SESSION_ID()

return value is string.

USER

Retrieve the name of the user executing the query.

USER([includeSecurityDomain])

includeSecurityDomain is a boolean. return value is string.

If includeSecurityDomain is omitted or true, then the user name will be returned with @security-domain appended.

CURRENT_DATABASE

Retrieve the catalog name of the database. The VDB name is always the catalog name.

CURRENT_DATABASE()

return value is string.

TEIID_SESSION_GET

Retrieve the session variable.

TEIID_SESSION_GET(name)

name is a string and the return value is an object.

A null name will return a null value. Typically you will use the a get wrapped in a CAST to convert to the desired type.

TEIID_SESSION_SET

Set the session variable.

TEIID_SESSION_SET(name, value)

name is a string, value is an object, and the return value is an object.

The previous value for the key or null will be returned. A set has no effect on the current transaction and is not affected by commit/rollback.

GENERATED_KEY

Get a column value from the generated keys of the last insert statement of this session returning a generated key.

Typically this function will only be used within the scope of procedure to determine a generated key value from an insert. Not all inserts provide generated keys, because not all sources return generated keys.

GENERATED_KEY()

The return value is long.

Returns the first column of the last generated key as a long value. Null is returned if there is no such generated key.

GENERATED_KEY(column_name)`

column_name is a string. The return value is of type object.

A more general form of **GENERATED_KEY** that can be used if there are more than one generated column or a type other than long. Null is returned if there is no such generated key nor matching key column.

3.5.9. XML functions

XML functions provide functionality for working with XML data. For more information, see *JSONTOXML* in JSON functions.

Sample data for examples

Examples provided with XML functions use the following table structure

```
TABLE  Customer (
   CustomerId integer PRIMARY KEY,
   CustomerName varchar(25),
   ContactName varchar(25)
   Address varchar(50),
   City varchar(25),
   PostalCode varchar(25),
   Country varchar(25),
);
```

with Data

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
87	Wartian Herkku	Pirkko Koskitalo	Torikatu 38	Oulu	90110	Finland
88	Wellington Importadora	Paula Parente	Rua do Mercado, 12	Resende	08737-363	Brazil
89	White Clover Markets	Karl Jablonski	305 - 14th Ave. S. Suite 3B	Seattle	98128	USA

XMLCAST

Cast to or from XML.

```
XMLCAST(expression AS type)
```

Expression or type must be XML. The return value will be typed as **type**. This is the same functionality that **XMLTABLE** uses to convert values to the desired runtime type, except that **XMLCAST** does not work with array type targets.

XMLCOMMENT

Returns an XML comment.

```
XMLCOMMENT(comment)
```

Comment is a string. Return value is XML.

XMLCONCAT

Returns an XML with the concatenation of the given XML types.

```
XMLCONCAT(content [, content]*)
```

Content is XML. Return value is XML.

If a value is null, it will be ignored. If all values are null, null is returned.

Concatenate two or more XML fragments

```
SELECT XMLCONCAT(
         XMLELEMENT("name", CustomerName),
         XMLPARSE(CONTENT '<a>b</a>' WELLFORMED)
       )
FROM   Customer c
WHERE  c.CustomerID = 87;

==========================================================
<name>Wartian Herkku</name><a>b</a>
```

XMLELEMENT

Returns an XML element with the given name and content.

```
XMLELEMENT([NAME] name [, <NSP>] [, <ATTR>][, content]*)

ATTR:=XMLATTRIBUTES(exp [AS name] [, exp [AS name]]*)

NSP:=XMLNAMESPACES((uri AS prefix | DEFAULT uri | NO DEFAULT))+
```

If the content value is of a type other than XML, it will be escaped when added to the parent element. Null content values are ignored. Whitespace in XML or the string values of the content is preserved, but no whitespace is added between content values.

XMLNAMESPACES is used provide namespace information. NO DEFAULT is equivalent to defining the default namespace to the null uri - xmlns="". Only one DEFAULT or NO DEFAULT namespace item may be specified. The namespace prefixes xmlns and xml are reserved.

If a attribute name is not supplied, the expression must be a column reference, in which case the attribute name will be the column name. Null attribute values are ignored.

Name, prefix are identifiers. uri is a string literal. content can be any type. Return value is XML. The return value is valid for use in places where a document is expected.

Simple example

```
SELECT XMLELEMENT("name", CustomerName)
FROM   Customer c
WHERE  c.CustomerID = 87;

==========================================================
<name>Wartian Herkku</name>
```

Multiple columns

```
SELECT XMLELEMENT("customer",
         XMLELEMENT("name", c.CustomerName),
         XMLELEMENT("contact", c.ContactName))
FROM   Customer c
WHERE  c.CustomerID = 87;

==========================================================
<customer><name>Wartian Herkku</name><contact>Pirkko Koskitalo</contact></customer>
```

Columns as attributes

```
SELECT XMLELEMENT("customer",
         XMLELEMENT("name", c.CustomerName,
           XMLATTRIBUTES(
               "contact" as c.ContactName,
               "id" as c.CustomerID
           )
         )
       )
FROM   Customer c
WHERE  c.CustomerID = 87;

==========================================================
<customer><name contact="Pirkko Koskitalo" id="87">Wartian Herkku</name></customer>
```

XMLFOREST

Returns an concatenation of XML elements for each content item.

```
XMLFOREST(content [AS name] [, <NSP>] [, content [AS name]]*)
```

For the definition of NSP - XMLNAMESPACES, see See *XMLELEMENT* in XML functions.

Name is an identifier. Content can be any type. Return value is XML.

If a name is not supplied for a content item, the expression must be a column reference, in which case the element name will be a partially escaped version of the column name.

You can use the XMLFOREST to simplify the declaration of multiple XMLELEMENTS. The XMLFOREST function allows you to process multiple columns at once.

Example

```
SELECT XMLELEMENT("customer",
         XMLFOREST(
           c.CustomerName AS "name",
           c.ContactName AS "contact"
         ))
FROM   Customer c
WHERE  c.CustomerID = 87;

==========================================================
<customer><name>Wartian Herkku</name><contact>Pirkko Koskitalo</contact></customer>
```

XMLAGG

XMLAGG is an aggregate function, that takes a collection of XML elements and returns an aggregated XML document.

```
XMLAGG(xml)
```

From above example in XMLElement, each row in the Customer table table will generate row of XML if there are multiple rows matching the criteria. That will generate a valid XML, but it will not be well formed, because it lacks the root element. XMLAGG can used to correct that

Example

```
SELECT XMLELEMENT("customers",
        XMLAGG(
          XMLELEMENT("customer",
           XMLFOREST(
             c.CustomerName AS "name",
             c.ContactName AS "contact"
           )))
FROM   Customer c


==========================================================
<customers>
<customer><name>Wartian Herkku</name><contact>Pirkko Koskitalo</contact></customer>
<customer><name>Wellington Importadora</name><contact>Paula Parente</contact></customer>
<customer><name>White Clover Markets</name><contact>Karl Jablonski</contact></customer>
</customers>
```

XMLPARSE

Returns an XML type representation of the string value expression.

```
XMLPARSE((DOCUMENT|CONTENT) expr [WELLFORMED])
```

expr in {string, clob, blob, varbinary}. Return value is XML.

If DOCUMENT is specified then the expression must have a single root element and may or may not contain an XML declaration.

If WELLFORMED is specified then validation is skipped; this is especially useful for CLOB and BLOB known to already be valid.

```
SELECT XMLPARSE(CONTENT '<customer><name>Wartian Herkku</name><contact>Pirkko
Koskitalo</contact></customer>' WELLFORMED);

Will return a SQLXML with contents
===============================================================
<customer><name>Wartian Herkku</name><contact>Pirkko Koskitalo</contact></customer>
```

XMLPI

Returns an XML processing instruction.

```
XMLPI([NAME] name [, content])
```

Name is an identifier. Content is a string. Return value is XML.

XMLQUERY

Returns the XML result from evaluating the given xquery.

```
XMLQUERY([<NSP>] xquery [<PASSING>] [(NULL|EMPTY) ON EMPTY]]

PASSING:=PASSING exp [AS name] [, exp [AS name]]*
```

For the definition of NSP - XMLNAMESPACES, see *XMLELEMENT* in XML functions.

Namespaces may also be directly declared in the xquery prolog.

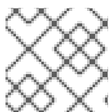
The optional PASSING clause is used to provide the context item, which does not have a name, and named global variable values. If the xquery uses a context item and none is provided, then an exception will be raised. Only one context item may be specified and should be an XML type. All non-context non-XML passing values will be converted to an appropriate XML type. Null will be returned if the context item evaluates to null.

The ON EMPTY clause is used to specify the result when the evaluted sequence is empty. EMPTY ON EMPTY, the default, returns an empty XML result. NULL ON EMPTY returns a null result.

xquery in string. Return value is XML.

XMLQUERY is part of the SQL/XML 2006 specification.

For more information, see XMLTABLE in FROM clause.



NOTE

See also XQuery optimization.

XMLEXISTS

Returns true if a non-empty sequence would be returned by evaluating the given xquery.

```
XMLEXISTS([<NSP>] xquery [<PASSING>]]

PASSING:=PASSING exp [AS name] [, exp [AS name]]*
```

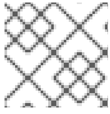
For the definition of NSP - XMLNAMESPACES, see *XMLELEMENT* in XML functions.

Namespaces can also be directly declared in the xquery prolog.

The optional PASSING clause is used to provide the context item, which does not have a name, and named global variable values. If the xquery uses a context item and none is provided, then an exception will be raised. Only one context item may be specified and should be an XML type. All non-context non-XML passing values will be converted to an appropriate XML type. Null/Unknown will be returned if the context item evaluates to null.

xquery in string. Return value is boolean.

XMLEXISTS is part of the SQL/XML 2006 specification.



NOTE

See also XQuery optimization.

XMLSERIALIZE

Returns a character type representation of the XML expression.

```
XMLSERIALIZE([(DOCUMENT|CONTENT)] xml [AS datatype] [ENCODING enc] [VERSION ver]
[(INCLUDING|EXCLUDING) XMLDECLARATION])
```

Return value matches datatype. If no datatype is specified, then clob will be assumed.

The type may be character (string, varchar, clob) or binary (blob, varbinar). CONTENT is the default. If DOCUMENT is specified and the XML is not a valid document or fragment, then an exception is raised.

The encoding enc is specified as an identifier. A character serialization may not specify an encoding. The version ver is specified as a string literal. If a particular XMLDECLARATION is not specified, then the result will have a declaration only if performing a non UTF-8/UTF-16, or non version 1.0 document serialization or the underlying XML has an declaration. If CONTENT is being serialized, then the declaration will be omitted if the value is not a document or element.

See the following example that produces a BLOB of XML in UTF-16 including the appropriate byte order mark of FE FF and XML declaration.

Sample Binary Serialization

```
XMLSERIALIZE(DOCUMENT value AS BLOB ENCODING "UTF-16" INCLUDING
XMLDECLARATION)
```

XMLTEXT

Returns XML text.

```
XMLTEXT(text)
```

text is a string. Return value is XML.

XSLTRANSFORM

Applies an XSL stylesheet to the given document.

```
XSLTRANSFORM(doc, xsl)
```

Doc, XSL in {string, clob, xml}. Return value is a clob.

If either argument is null, the result is null.

XPATHVALUE

Applies the XPATH expression to the document and returns a string value for the first matching result. For more control over the results and XQuery, use the XMLQUERY function. For more information, see *XMLQUERY* in XML functions.

```
XPATHVALUE(doc, xpath)
```

Doc in {string, clob, blob, xml}. xpath is string. Return value is a string.

Matching a non-text node will still produce a string result, which includes all descendant text nodes. If a single element is matched that is marked with xsi:nil, then null will be returned.

When the input document utilizes namespaces, it is sometimes necessary to specify XPATH that ignores namespaces:

Sample XML for xpathValue Ignoring Namespaces

```
<?xml version="1.0" ?>
  <ns1:return xmlns:ns1="http://com.test.ws/exampleWebService">Hello<x> World</x></return>
```

Function:

Sample xpathValue Ignoring Namespaces

```
xpathValue(value, '/*[local-name()="return"]')
```

Results in **Hello World**

Example: Generating hierarchical XML from flat data structure

With following table and its contents

```
Table {
 x string,
 y integer
}
```

data like ['a', 1], ['a', 2], ['b', 3], ['b', 4], if you want generate a XML that looks like

```
<root>
   <x>
       a
       <y>1</y>
       <y>2</y>
   </x>
   <x>
       b
       <y>3</y>
       <y>4</y>
   </x>
</root>
```

use the SQL statement in Data Virtualization as below

```
select xmlelement(name "root", xmlagg(p))
   from (select xmlelement(name "x", x, xmlagg(xmlelement(name "y", y)) as p from tbl group by x)) as
v
```

For more examples, see http://oracle-base.com/articles/misc/sqlxml-sqlx-generating-xml-content-using-sql.php

3.5.10. JSON functions

JSON functions provide functionality for working with JSON (JavaScript Object Notation) data.

Sample data for examples

Examples provided with XML functions use the following table structure:

```
TABLE  Customer (
    CustomerId integer PRIMARY KEY,
    CustomerName varchar(25),
    ContactName varchar(25)
    Address varchar(50),
    City varchar(25),
    PostalCode varchar(25),
    Country varchar(25),
);
```

with Data

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
87	Wartian Herkku	Pirkko Koskitalo	Torikatu 38	Oulu	90110	Finland
88	Wellington Importadora	Paula Parente	Rua do Mercado, 12	Resende	08737-363	Brazil
89	White Clover Markets	Karl Jablonski	305 - 14th Ave. S. Suite 3B	Seattle	98128	USA

JSONARRAY

Returns a JSON array.

```
JSONARRAY(value...)
```

value is any object that can be converted to a JSON value. For more information, see JSON functions. Return value is JSON.

Null values will be included in the result as null literals.

mixed value example

```
jsonArray('a"b', 1, null, false, {d'2010-11-21'})
```

Would return

```
["a\"b",1,null,false,"2010-11-21"]
```

Using JSONARRAY on a Table

```
SELECT JSONARRAY(CustomerId, CustomerName)
FROM   Customer c
WHERE  c.CustomerID >= 88;
==========================================================
[88,"Wellington Importadora"]
[89,"White Clover Markets"]
```

JSONOBJECT

Returns a JSON object.

```
JSONARRAY(value [as name] ...)
```

value is any object that can be converted to a JSON value. For more information, see JSON functions. Return value is JSON.

Null values will be included in the result as null literals.

If a name is not supplied and the expression is a column reference, the column name will be used otherwise exprN will be used where N is the 1-based index of the value in the JSONARRAY expression.

mixed value example

```
jsonObject('a"b' as val, 1, null as "null")
```

Would return

```
{"val":"a\"b","expr2":1,"null":null}
```

Using JSONOBJECT on a Table

```
SELECT JSONOBJECT(CustomerId, CustomerName)
FROM   Customer c
WHERE  c.CustomerID >= 88;
==========================================================
{"CustomerId":88, "CustomerName":"Wellington Importadora"}
{"CustomerId":89, "CustomerName":"White Clover Markets"}
```

Another example

```
SELECT JSONOBJECT(JSONOBJECT(CustomerId, CustomerName) as Customer)
FROM   Customer c
WHERE  c.CustomerID >= 88;
==========================================================
{"Customer":{"CustomerId":88, "CustomerName":"Wellington Importadora"}}
{"Customer":{"CustomerId":89, "CustomerName":"White Clover Markets"}}
```

Another example

```
SELECT JSONOBJECT(JSONARRAY(CustomerId, CustomerName) as Customer)
FROM   Customer c
WHERE  c.CustomerID >= 88;
```

```
==========================================================
{"Customer":[88, "Wellington Importadora"]}
{"Customer":[89, "White Clover Markets"]}
```

JSONPARSE

Validates and returns a JSON result.

```
JSONPARSE(value, wellformed)
```

value is blob with an appropriate JSON binary encoding (UTF-8, UTF-16, or UTF-32) or a clob. wellformed is a boolean indicating that validation should be skipped. Return value is JSON.

A null for either input will return null.

JSON parse of a simple literal value

```
jsonParse('{"Customer":{"CustomerId":88, "CustomerName":"Wellington Importadora"}}', true)
```

JSONARRAY_AGG

creates a JSON array result as a Clob including null value. This is similar to JSONARRAY but aggregates its contents into single object

```
SELECT JSONARRAY_AGG(JSONOBJECT(CustomerId, CustomerName))
FROM   Customer c
WHERE  c.CustomerID >= 88;
==========================================================
[{"CustomerId":88, "CustomerName":"Wellington Importadora"}, {"CustomerId":89,
"CustomerName":"White Clover Markets"}]
```

You can also wrap array as

```
SELECT JSONOBJECT(JSONARRAY_AGG(JSONOBJECT(CustomerId as id, CustomerName as
name)) as Customer)
FROM   Customer c
WHERE  c.CustomerID >= 88;
==========================================================
{"Customer":[{"id":89,"name":"Wellington Importadora"},{"id":100,"name":"White Clover Markets"}]}
```

Conversion to JSON

A straight-forward, specification-compliant conversion is used for converting values into their appropriate JSON document form.

- Null values are included as the null literal.
- Values parsed as JSON or returned from a JSON construction function (JSONPARSE, JSONARRAY, JSONARRAY_AGG) will be directly appended into a JSON result.
- Boolean values are included as true/false literals.
- Numeric values are included as their default string conversion - in some circumstances if not a number or +-infinity results are allowed, invalid JSON may be obtained.

- String values are included in their escaped/quoted form.
- Binary values are not implicitly convertable to JSON values and require a specific prior to inclusion in JSON.
- All other values will be included as their string conversion in the appropriate escaped/quoted form.

JSONTOXML

Returns an XML document from JSON.

```
JSONTOXML(rootElementName, json)
```

rootElementName is a string, **json** is in {clob, blob}. Return value is XML.

The appropriate UTF encoding (8, 16LE. 16BE, 32LE, 32BE) will be detected for JSON blobs. If another encoding is used, see the TO_CHARS function in String functions.

The result is always a well-formed XML document.

The mapping to XML uses the following rules:

- The current element name is initially the rootElementName, and becomes the object value name as the JSON structure is traversed.
- All element names must be valid XML 1.1 names. Invalid names are fully escaped according to the SQLXML specification.
- Each object or primitive value will be enclosed in an element with the current name.
- Unless an array value is the root, it will not be enclosed in an additional element.
- Null values will be represented by an empty element with the attribute xsi:nil="true"
- Boolean and numerical value elements will have the attribute xsi:type set to boolean and decimal respectively.

JSON:

Sample JSON to XML for jsonToXml('person', x)

```
{"firstName" : "John" , "children" : [ "Randy", "Judy" ]}
```

XML:

Sample JSON to XML for jsonToXml('person', x)

```
<?xml version="1.0" ?>
  <person>
    <firstName>John</firstName>
    <children>Randy</children>
    <children>Judy<children>
  </person>
```

JSON:

Sample JSON to XML for jsonToXml('person', x) with a root array

```
[{"firstName" : "George" }, { "firstName" : "Jerry" }]
```

XML (Notice there is an extra "person" wrapping element to keep the XML well-formed):

Sample JSON to XML for jsonToXml('person', x) with a root array

```
<?xml version="1.0" ?>
<person>
  <person>
    <firstName>George</firstName>
  </person>
  <person>
    <firstName>Jerry</firstName>
  </person>
</person>
```

JSON:

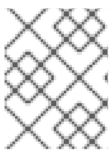
Sample JSON to XML for jsonToXml('root', x) with an invalid name

```
{"/invalid" : "abc" }
```

XML:

Sample JSON to XML for jsonToXml('root', x) with an invalid name

```
<?xml version="1.0" ?>
<root>
  <_x002F_invalid>abc</_x002F_invalid>
</root>
```



NOTE

prior releases defaulted incorrectly to using *uXXXX* escaping rather than *xXXXX*. If you need to rely on that behavior see the org.teiid.useXMLxEscape system property.

JsonPath

Processing of JsonPath expressions is provided by Jayway JsonPath. Please note that it uses 0-based indexing, rather than 1-based indexing. Be sure that you are familiar with the expected returns for various path expressions. For example, if a row JsonPath expression is expected to provide an array, make sure that it's the array that you want, and not an array that would be returned automatically by an indefinite path expression.

If you encounter a situation where path names use reserved characters, such as '.', then you must use the bracketed JsonPath notation as that allows for any key, e.g. $['.key'].

For more information, see JSONTABLE.

JSONPATHVALUE

Extracts a single JSON value as a string.

```
JSONPATHVALUE(value, path [, nullLeafOnMissing])
```

value is a clob JSON document, **path** is a JsonPath string, and **nullLeafOnMissing** is a Boolean. Return value is a string value of the resulting JSON.

If **nullLeafOnMissing** is false (the default), then a path that evaluates to a leaf that is missing will throw an exception. If **nullLeafOnMissing** is true, then a null value will be returned.

If the value is an array produced by an indefinite path expression, then only the first value will be returned.

```
jsonPathValue('{"key":"value"}' '$.missing', true)
```

Would return

```
null
```

```
jsonPathValue('[{"key":"value1"}, {"key":"value2"}]' '$..key')
```

Would return

```
value1
```

JSONQUERY

Evaluate a JsonPath expression against a JSON document and return the JSON result.

```
JSONQUERY(value, path [, nullLeafOnMissing])
```

value is a clob JSON document, **path** is a JsonPath string, and **nullLeafOnMissing** is a Boolean. Return value is a JSON value.

If **nullLeafOnMissing** is false (the default), then a path that evaluates to a leaf that is missing will throw an exception. If **nullLeafOnMissing** is true, then a null value will be returned.

```
jsonPathValue('[{"key":"value1"}, {"key":"value2"}]' '$..key')
```

Would return

```
["value1","value2"]
```

3.5.11. Security functions

Security functions provide the ability to interact with the security system or to hash/encrypt values.

HASROLE

Whether the current caller has the Data Virtualization data role **roleName**.

```
hasRole([roleType,] roleName)
```

roleName must be a string, the return type is Boolean.

The two argument form is provided for backwards compatibility. **roleType** is a string and must be `data'.

Role names are case-sensitive and only match Data Virtualization Data roles. Foreign/JAAS roles/groups names are not valid for this function, unless there is corresponding data role with the same name.

MD5

Computes the MD5 hash of the value.

```
MD5(value)
```

value must be a string or varbinary, the return type is varbinary. String values are first converted to their UTF-8 byte representation.

SHA1

Computes the SHA-1 hash of the value.

```
SHA1(value)
```

value must be a string or varbinary, the return type is varbinary. String values are first converted to their UTF-8 byte representation.

SHA2_256

Computes the SHA-2 256 bit hash of the value.

```
SHA2_256(value)
```

value must be a string or varbinary, the return type is varbinary. String values are first converted to their UTF-8 byte representation.

SHA2_512

Computes the SHA-2 512 bit hash of the value.

```
SHA2_512(value)
```

value must be a string or varbinary, the return type is varbinary. String values are first converted to their UTF-8 byte representation.

AES_ENCRYPT

```
aes_encrypt(data, key)
```

AES_ENCRYPT() allow encryption of data using the official AES (Advanced Encryption Standard) algorithm, 16 bytes(128 bit) key length, and AES/CBC/PKCS5Padding cipher algorithm with an explicit initialization vector.

The **AES_ENCRYPT()** will return a BinaryType encrypted data. The argument **data** is the BinaryType data to encrypt, and the argument **key** is a BinaryType used in encryption.

AES_DECRYPT

```
aes_decrypt(data, key)
```

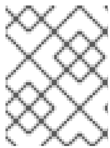
AES_DECRYPT() allow decryption of data using the official AES (Advanced Encryption Standard) algorithm, 16 bytes(128 bit) key length, and AES/CBC/PKCS5Padding cipher algorithm expecting an explicit initialization vector.

The **AES_DECRYPT()** will return a BinaryType decrypted data. The argument **data** is the BinaryType data to decrypt, and the argument **key** is a BinaryType used in decryption.

3.5.12. Spatial functions

Spatial functions provide functionality for working with geospatial data. Data Virtualization relies on the JTS Topology Suite to provide partial compatibility with the OpenGIS Simple Features Specification For SQL Revision 1.1. For more information about particular functions, see the Open GIS specification or the PostGIS manual.

Most Geometry capabilities is limited to two dimensions due to the WKB and WKT formats.



NOTE

There might be minor differences between Data Virtualization and pushdown results that will need to be further refined.

ST_GeomFromText

Returns a geometry from a Clob in WKT format.

```
ST_GeomFromText(text [, srid])
```

text is a CLOB, **srid** is an optional integer that represents a spatial reference identifier (SRID). Return value is a geometry.

ST_GeogFromText

Returns a geography from a Clob in (E)WKT format.

```
ST_GeogFromText(text)
```

text is a CLOB, **srid** is an optional integer. Return value is a geography.

ST_GeomFromWKB/ST_GeomFromBinary

Returns a geometry from a BLOB in WKB format.

```
ST_GeomFromWKB(bin [, srid])
```

bin is a BLOB, **srid** is an optional integer. Return value is a geometry.

ST_GeomFromEWKB

Returns a geometry from a BLOB in EWKB format.

```
ST_GeomFromEWKB(bin)
```

bin is a BLOB. Return value is a geometry. This version of the translator works with two dimensions only.

ST_GeogFromWKB

Returns a geography from a BLOB in (E)WKB format.

```
ST_GeomFromEWKB(bin)
```

bin is a BLOB. Return value is a geography. This version of the translator works with two dimensions only.

ST_GeomFromEWKT

Returns a geometry from a character large object (CLOB) in EWKT format.

```
ST_GeomFromEWKT(text)
```

text is a CLOB. Return value is a geometry. This version of the translator works with two dimensions only.

ST_GeomFromGeoJSON

Returns a geometry from a CLOB in GeoJSON format.

```
ST_GeomFromGeoJson(`text` [, srid])
```

text is a CLOB, **srid** is an optional integer. Return value is a geometry.

ST_GeomFromGML

Returns a geometry from a CLOB in GML2 format.

```
ST_GeomFromGML(text [, srid])
```

text is a CLOB, **srid** is an optional integer. Return value is a geometry.

ST_AsText

```
ST_AsText(geom)
```

geom is a geometry. Return value is CLOB in WKT format.

ST_AsBinary

```
ST_AsBinary(geo)
```

geo is a geometry or geography. Return value is a binary large object (BLOB) in WKB format.

ST_AsEWKB

```
ST_AsEWKB(geom)
```

geom is a geometry. Return value is BLOB in EWKB format.

ST_AsGeoJSON

```
ST_AsGeoJSON(geom)
```

geom is a geometry. Return value is a CLOB with the GeoJSON value.

ST_AsGML

```
ST_AsGML(geom)
```

geom is a geometry. Return value is a CLOB with the GML2 value.

ST_AsEWKT

```
ST_AsEWKT(geo)
```

geo is a geometry or geography. Return value is a CLOB with the EWKT value. The EWKT value is the WKT value with the SRID prefix.

ST_AsKML

```
ST_AsKML(geom)
```

geom is a geometry. Return value is a CLOB with the KML value. The KML value is effectively a simplified GML value and projected into SRID 4326.

&&

Returns true if the bounding boxes of **geom1** and **geom2** intersect.

```
geom1 && geom2
```

geom1, **geom2** are geometries. Return value is a Boolean.

ST_Contains

Returns true if **geom1** contains **geom2**.

```
ST_Contains(geom1, geom2)
```

geom1, **geom2** are geometries. Return value is a Boolean.

ST_Crosses

Returns true if the geometries cross.

```
ST_Crosses(geom1, geom2)
```

geom1, **geom2** are geometries. Return value is a Boolean.

ST_Disjoint

Returns true if the geometries are disjoint.

```
ST_Disjoint(geom1, geom2)
```

geom1, **geom2** are geometries. Return value is a Boolean.

ST_Distance

Returns the distance between two geometries.

```
ST_Distance(geo1, geo2)
```

geo1, **geo2** are both geometries or geographies. Return value is a double. The geography variant must be pushed down for evaluation.

ST_DWithin

Returns true if the geometries are within a given distance of one another.

```
ST_DWithin(geom1, geom2, dist)
```

geom1, **geom2** are geometries. **dist** is a double. Return value is a Boolean.

ST_Equals

Returns true if the two geometries are spatially equal. The points and order can differ, but neither geometry lies outside of the other.

```
ST_Equals(geom1, geom2)
```

geom1, **geom2** are geometries. Return value is a Boolean.

ST_Intersects

Returns true if the geometries intersect.

```
ST_Intersects(geo1, geo2)
```

geo1, **geo2** are both geometries or geographies. Return value is a Boolean. The geography variant must be pushed down for evaluation.

ST_OrderingEquals

Returns true if **geom1** and **geom2** have the same structure and the same ordering of points.

```
ST_OrderingEquals(geom1, geom2)
```

geom1, **geom2** are geometries. Return value is a Boolean.

ST_Overlaps

Returns true if the geometries overlap.

```
ST_Overlaps(geom1, geom2)
```

geom1, **geom2** are geometries. Return value is a Boolean.

ST_Relate

Test or return the intersection of geom1 and geom2.

```
ST_Relate(geom1, geom2, pattern)
```

geom1, **geom2** are geometries. **pattern** is a nine character DE-9IM pattern string. Return value is a Boolean.

```
ST_Relate(geom1, geom2)
```

geom1, **geom2** are geometries. Return value is the nine character DE-9IM intersection string.

ST_Touches

Returns true if the geometries touch.

```
ST_Touches(geom1, geom2)
```

geom1, **geom2** are geometries. Return value is a Boolean.

ST_Within

Returns true if **geom1** is completely inside **geom2**.

```
ST_Within(geom1, geom2)
```

geom1, **geom2** are geometries. Return value is a Boolean.

ST_Area

Returns the area of geom.

```
ST_Area(geom)
```

geom is a geometry. Return value is a double.

ST_CoordDim

Returns the coordinate dimensions of geom.

```
ST_CoordDim(geom)
```

geom is a geometry. Return value is an integer between 0 and 3.

ST_Dimension

Returns the dimension of geom.

```
ST_Dimension(geom)
```

geom is a geometry. Return value is an integer between 0 and 3.

ST_EndPoint

Returns the end Point of the LineString geom. Returns null if **geom** is not a LineString.

```
ST_EndPoint(geom)
```

geom is a geometry. Return value is a geometry.

ST_ExteriorRing

Returns the exterior ring or shell LineString of the polygon geom. Returns null if **geom** is not a polygon.

```
ST_ExteriorRing(geom)
```

geom is a geometry. Return value is a geometry.

ST_GeometryN

Returns the nth geometry at the given 1-based index in geom. Returns null if a geometry at the given index does not exist. Non-collection types return themselves at the first index.

```
ST_GeometryN(geom, index)
```

geom is a geometry. index is an integer. Return value is a geometry.

ST_GeometryType

Returns the type name of **geom** as ST_name. Where name will be LineString, Polygon, Point etc.

```
ST_GeometryType(geom)
```

geom is a geometry. Return value is a string.

ST_HasArc

Tests if the geometry has a circular string. Reports **false**, because the translator does not work with curved geometry types.

```
ST_HasArc(geom)
```

geom is a geometry. Return value is a geometry.

ST_InteriorRingN

Returns the nth interior ring LinearString geometry at the given 1-based index in geom. Returns null if a geometry at the given index does not exist, or if **geom** is not a polygon.

```
ST_InteriorRingN(geom, index)
```

geom is a geometry. index is an integer. Return value is a geometry.

ST_IsClosed

Returns true if LineString **geom** is closed. Returns false if **geom** is not a LineString

```
ST_IsClosed(geom)
```

geom is a geometry. Return value is a Boolean.

ST_IsEmpty

Returns true if the set of points is empty.

```
ST_IsEmpty(geom)
```

geom is a geometry. Return value is a Boolean.

ST_IsRing

Returns true if the LineString **geom** is a ring. Returns false if **geom** is not a LineString.

```
ST_IsRing(geom)
```

geom is a geometry. Return value is a Boolean.

ST_IsSimple

Returns true if the **geom** is simple.

```
ST_IsSimple(geom)
```

geom is a geometry. Return value is a Boolean.

ST_IsValid

Returns **true** if the **geom** is valid.

```
ST_IsValid(geom)
```

geom is a geometry. Return value is a Boolean.

ST_Length

Returns the length of a (Multi)LineString, otherwise returns 0.

```
ST_Length(geo)
```

geo is a geometry or a geography. Return value is a double. The geography variant must be pushed down for evaluation.

ST_NumGeometries

Returns the number of geometries in **geom**. Will return 1 if not a geometry collection.

```
ST_NumGeometries(geom)
```

geom is a geometry. Return value is an integer.

ST_NumInteriorRings

Returns the number of interior rings in the polygon geometry. Returns null if **geom** is not a polygon.

```
ST_NumInteriorRings(geom)
```

geom is a geometry. Return value is an integer.

ST_NunPoints

Returns the number of points in **geom**.

```
ST_NunPoints(geom)
```

geom is a geometry. Return value is an integer.

ST_PointOnSurface

Returns a point that is guaranteed to be on the surface of geom.

```
ST_PointOnSurface(geom)
```

geom is a geometry. Return value is a point geometry.

ST_Perimeter

Returns the perimeter of the (Multi)Polygon geom. Will return 0 if **geom** is not a (Multi)Polygon

```
ST_Perimeter(geom)
```

geom is a geometry. Return value is a double.

ST_PointN

Returns the nth point at the given 1-based index in geom. Returns null if a point at the given index does not exist or if **geom** is not a LineString.

```
ST_PointN(geom, index)
```

geom is a geometry. index is an integer. Return value is a geometry.

ST_SRID

Returns the SRID for the geometry.

```
ST_SRID(geo)
```

geo is a geometry or geography. Return value is an integer. A 0 value rather than null will be returned for an unknown SRID on a non-null geometry.

ST_SetSRID

Set the SRID for the given geometry.

```
ST_SetSRID(geo, srid)
```

geo is a geometry or geography. **srid** is an integer. Return value is the same as the value of **geo**. Only the SRID metadata of is modified. No transformation is performed.

ST_StartPoint

Returns the start Point of the LineString geom. Returns null if **geom** is not a LineString.

```
ST_StartPoint(geom)
```

geom is a geometry. Return value is a geometry.

ST_X

Returns the X ordinate value, or null if the point is empty. Throws an exception if the geometry is not a point.

```
ST_X(geom)
```

geom is a geometry. Return value is a double.

ST_Y

Returns the Y ordinate value, or null if the point is empty. Throws an exception if the geometry is not a point.

```
ST_Y(geom)
```

geom is a geometry. Return value is a double.

ST_Z

Returns the Z ordinate value, or null if the point is empty. Throws an exception if the geometry is not a point. Typically returns **null** because the translator does not work with more than two dimensions.

```
ST_Z(geom)
```

geom is a geometry. Return value is a double.

ST_Boundary

Computes the boundary of the given geometry.

```
ST_Boundary(geom)
```

geom is a geometry. Return value is a geometry.

ST_Buffer

Computes the geometry that has points within the given distance of **geom**.

```
ST_Buffer(geom, distance)
```

geom is a geometry. **distance** is a double. Return value is a geometry.

ST_Centroid

Computes the geometric center point of geom.

```
ST_Centroid(geom)
```

geom is a geometry. Return value is a geometry.

ST_ConvexHull

Return the smallest convex polygon that contains all of the points in geometry.

```
ST_ConvexHull(geom)
```

geom is a geometry. Return value is a geometry.

ST_CurveToLine

Converts a CircularString/CurvedPolygon to a LineString/Polygon. Not currently implemented in Data Virtualization.

```
ST_CurveToLine(geom)
```

geom is a geometry. Return value is a geometry.

ST_Difference

Computes the closure of the point set of the points contained in **geom1** that are not in **geom2**.

```
ST_Difference(geom1, geom2)
```

geom1, **geom2** are geometries. Return value is a geometry.

ST_Envelope

Computes the 2D bounding box of the given geometry.

```
ST_Envelope(geom)
```

geom is a geometry. Return value is a geometry.

ST_Force_2D

Removes the z coordinate value if present.

```
ST_Force_2D(geom)
```

geom is a geometry. Return value is a geometry.

ST_Intersection

Computes the point set intersection of the points contained in **geom1** and in **geom2**.

```
ST_Intersection(geom1, geom2)
```

geom1, **geom2** are geometries. Return value is a geometry.

ST_Simplify

Simplifies a geometry using the Douglas-Peucker algorithm, but may oversimplify to an invalid or empty geometry.

```
ST_Simplify(geom, distanceTolerance)
```

geom is a geometry. **distanceTolerance** is a double. Return value is a geometry.

ST_SimplifyPreserveTopology

Simplifies a geometry using the Douglas-Peucker algorithm. Will always return a valid geometry.

```
ST_SimplifyPreserveTopology(geom, distanceTolerance)
```

geom is a geometry. **distanceTolerance** is a double. Return value is a geometry.

ST_SnapToGrid

Snaps all points in the geometry to grid of given size.

```
ST_SnapToGrid(geom, size)
```

geom is a geometry. size is a double. Return value is a geometry.

ST_SymDifference

Return the part of geom1 that does not intersect with geom2 and vice versa.

```
ST_SymDifference(geom1, geom2)
```

geom1, **geom2** are geometry. Return value is a geometry.

ST_Transform

Transforms the geometry value from one coordinate system to another.

```
ST_Transform(geom, srid)
```

geom is a geometry. **srid** is an integer. Return value is a geometry. The **srid** value and the SRID of the geometry value must exist in the SPATIAL_REF_SYS view.

ST_Union

Return a geometry that represents the point set containing all of **geom1** and **geom2**.

```
ST_Union(geom1, geom2)
```

geom1, **geom2** are geometries. Return value is a geometry.

ST_Extent

Computes the 2D bounding box around all of the geometry values. All values should have the same SRID.

```
ST_Extent(geom)
```

geom is a geometry. Return value is a geometry.

ST_Point

Retuns the Point for the given coordinates.

```
ST_Point(x, y)
```

x and y are doubles. Return value is a Point geometry.

ST_Polygon

Returns the Polygon with the given shell and SRID.

```
ST_Polygon(geom, srid)
```

geom is a linear ring geometry and **srid** is an integer. Return value is a Polygon geometry.

3.5.13. Miscellaneous functions

Documents additional functions and those contributed by other projects.

array_get

Returns the object value at a given array index.

```
array_get(array, index)
```

array is the object type, **index** must be an integer, and the return type is an object.

1-based indexing is used. The actual array value should be a java.sql.Array or java array type. A null is returned if either argument is null, or if the index is out of bounds.

array_length

Returns the length for a given array.

```
array_length(array)
```

array is the object type, and the return type is integer.

The actual array value should be a java.sql.Array or java array type. An exception is thrown if the array value is the wrong type.

uuid

Returns a universally unique identifier.

```
uuid()
```

The return type is string.

Generates a type 4 (pseudo randomly generated) UUID using a cryptographically strong random number generator. The format is XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX where each X is a hex digit.

Data quality functions

Data Quality functions are contributed by the ODDQ Project. The functions are prefixed with **osdq.**, but can be called without the prefix.

osdq.random

Returns the randomized string. For example, **jboss teiid** may randomize to **jtids soibe**.

```
random(sourceValue)
```

The **sourceValue** is the string to be randomized.

osdq.digit

Returns digit characters of the string. For example, **a1 b2 c3 d4** becomes **1234**.

```
digit(sourceValue)
```

The **sourceValue** is the string from which you want to extract digit characters.

osdq.whitespaceIndex

Returns the index of the first whitespace. For example, **jboss teiid** will return **5**.

```
whitespaceIndex(sourceValue)
```

The sourceValue is the string from which you want to find the whitespace index.

osdq.validCreditCard

Check whether a credit card number is valid. Returns **true** if it matches credit card logic and checksum.

```
validCreditCard(cc)
```

cc is the credit card number string to check.

osdq.validSSN

Check whether a social security number (SSN) is valid. Returns **true** if it matches SSN logic.

```
validSSN(ssn)
```

ssn is the social security number string to check.

osdq.validPhone

Check whether a phone number is valid. Returns **true** if the number matches phone logic. Numbers must contain more than 8, but less than 12 characters, and cannot start with **000**.

```
validPhone(phone)
```

`**phone** is the phone number string need to check.

osdq.validEmail

Check whether an email address is valid. Returns **true** if valid.

```
validEmail(email)
```

email is the email address string to check.

osdq.cosineDistance

Returns the float distance between two strings based on the Cosine Similarity algorithm.

```
cosineDistance(a, b)
```

a and **b** are strings for which you want to calculate the distance.

osdq.jaccardDistance

Returns the float distance between two strings, based on the Jaccard similarity algorithm.

```
jaccardDistance(a, b)
```

The **a** and **b** are strings for which you want to calculate the distance.

osdq.jaroWinklerDistance

Returns the float distance between two strings based on the Jaro-Winkler algorithm.

```
jaroWinklerDistance(a, b)
```

The **a** and **b** are strings for which you want to calculate the distance.

osdq.levenshteinDistance

Returns the float distance between two strings based on the Levenshtein algorithm.

```
levenshteinDistance(a, b)
```

The **a** and **b** are strings for which you want to calculate the distance.

osdq.intersectionFuzzy

Returns the set of unique elements from the first set with cosine distance less than the specified value to every member of the second set.

```
intersectionFuzzy(a, b)
```

a and **b** are string arrays. **c** is a float representing the distance, such that 0.0 or less will match any and > 1.0 will match exact.

osdq.minusFuzzy

Returns the set of unique elements from the first set with cosine distance less than the specified value to every member of the second set.

```
minusFuzzy(a, b, c)
```

a and **b** are string arrays. **c** is a float representing the distance, such that 0.0 or less will match any and > 1.0 will match exact.

osdq.unionFuzzy

Returns the set of unique elements that contains members from the first set and members of the second set that have a cosine distance less than the specified value to every member of the first set.

```
unionFuzzy(a, b, c)
```

a and **b** are string arrays. **c** is a float representing the distance, such that 0.0 or less will match any and > 1.0 will match exact.

3.5.14. Nondeterministic function handling

Data Virtualization categorizes functions by varying degrees of determinism. When a function is evaluated and to what extent the result can be cached are based upon its determinism level.

Deterministic

The function always returns the same result for the given inputs. Deterministic functions are evaluated by the engine as soon as all input values are known, which may occur as soon as the rewrite phase. Some functions, such as the lookup function, are not truly deterministic, but are treated as such for performance. All functions that are not categorized according to the remaining items in this list are considered deterministic.

User Deterministic

The function returns the same result for the given inputs for the same user. This includes the **hasRole** and **user** functions. User deterministic functions are evaluated by the engine as soon as all input values are known, which may occur as soon as the rewrite phase. If a user deterministic function is evaluated during the creation of a prepared processing plan, then the resulting plan will be cached only for the user.

Session Deterministic

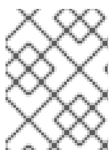
The function returns the same result for the given inputs under the same user session. This category includes the **env** function. Session deterministic functions are evaluated by the engine as soon as all input values are known, which may occur as soon as the rewrite phase. If a session deterministic function is evaluated during the creation of a prepared processing plan, then the resulting plan will be cached only for the user's session.

Command Deterministic

The result of function evaluation is only deterministic within the scope of the user command. This category include the **curdate**, **curtime**, **now**, and **commandpayload** functions. Command deterministic functions are delayed in evaluation until processing to ensure that even prepared plans utilizing these functions will be executed with relevant values. Command deterministic function evaluation will occur prior to pushdown. However, multiple occurrences of the same command deterministic time function are not guaranteed to evaluate to the same value.

Nondeterministic

The result of function evaluation is fully nondeterministic. This category includes the **rand** function and UDFs marked as **nondeterministic**. Nondeterministic functions are delayed in evaluation until processing with a preference for pushdown. If the function is not pushed down, then it may be evaluated for every row in it's execution context (for example, if the function is used in the select clause).



NOTE

Uncorrelated subqueries will be treated as deterministic regardless of the functions used within them.

3.6. DML COMMANDS

You can use SQL in Data Virtualization to issue queries and define view transformations. For more information about how SQL is used in virtual procedures and update procedures, see Procedure language. Nearly all these features follow standard SQL syntax and functionality, so you can use any SQL reference for more information.

There are 4 basic commands for manipulating data in SQL, corresponding to the create, read, update, and delete (CRUD) operations: INSERT, SELECT, UPDATE, and DELETE. A MERGE statement acts as a combination of INSERT and UPDATE.

You can also execute procedures by using the EXECUTE command, procedural relational command. For more information, see Procedural relational command, or Anonymous procedure block.

3.6.1. Set operations

You can use the SQL **UNION**, **UNION ALL**, **INTERSECT**, and **EXCEPT** set operations in Data Virtualization to combine the results of query expressions.

Usage:

```
queryExpression (UNION|INTERSECT|EXCEPT) [ALL] queryExpression [ORDER BY...]
```

Syntax Rules:

- The output columns will be named by the output columns of the first set operation branch.
- Each **SELECT** must have the same number of output columns and compatible data types for each relative column. Data type conversion is performed if data types are inconsistent and implicit conversions exist.
- If **UNION**, **INTERSECT**, or **EXCEPT** is specified without **all**, then the output columns must be comparable types.
- You cannot use the SQL **INTERSECT ALL** or **EXCEPT ALL** operators.

3.6.2. SELECT command

The SELECT command is used to retrieve records for any number of relations.

A SELECT command can contain the following clauses:

- WITH ...
- SELECT ...
- FROM ...
- WHERE ...
- GROUP BY ...
- HAVING ...
- ORDER BY ...
- (LIMIT ...) | ([OFFSET ...] [FETCH ...])
- OPTION ...

Except for the OPTION clause, all of the preceding clauses are defined by the SQL specification. The specification also specifies the order in which these clauses are logically processed. Processing occurs in stages, with each stage passing a set of rows to the following stage. The processing model is logical, and does not represent the way that a database engine performs the processing, but it is a useful model for understanding how SQL works. The SELECT command processes clauses in the following stages:

Stage 1: WITH clause

Gathers all rows from all with items in the order listed. Subsequent WITH items and the main query can reference a WITH item as if it were a table.

Stage 2: FROM clause

Gathers all rows from all tables involved in the query and logically joins them with a Cartesian product to produce a single large table with all columns from all tables. Joins and join criteria are then applied to filter rows that do not match the join structure.

Stage 3: WHERE clause

Applies a criteria to every output row from the FROM stage, further reducing the number of rows.

Stage 4: GROUP BY clause

Groups sets of rows with matching values in the GROUP BY columns.

Stage 5: HAVING clause

Applies criteria to each group of rows. Criteria can only be applied to columns that will have constant values within a group (those in the grouping columns or aggregate functions applied across the group).

Stage 6: SELECT clause

Specifies the column expressions that should be returned from the query. Expressions are evaluated, including aggregate functions that are based on the groups of rows, which will no longer exist after this point. The output columns are named using either column aliases or an implicit name determined by the engine. If SELECT DISTINCT is specified, duplicate removal is performed on the rows being returned from the SELECT stage.

Stage 7: ORDER BY clause

Sorts the rows returned from the SELECT stage as desired. Supports sorting on multiple columns in specified order, ascending or descending. The output columns will be identical to those columns returned from the SELECT stage and will have the same name.

Stage 8: LIMIT clause

Returns only the specified rows (with skip and limit values).

The preceding model helps to understand how SQL works. For example, given that the SELECT clause assigns aliases to columns, it makes sense that the subsequent ORDER BY clause must use those aliases to reference columns. Without knowledge of the processing model, this can be somewhat confusing. Seen in light of the model, it is clear that the ORDER BY stage is the only stage occurring after the SELECT stage, which is where the columns are named. Because the WHERE clause is processed before the SELECT, the columns have not yet been named and the aliases are not yet known.

TIP

The explicit table syntax **TABLE x** may be used as a shortcut for **SELECT * FROM x**.

3.6.3. VALUES command

The VALUES command is used to construct a simple table.

Example syntax

```
VALUES (value,...)
```

```
VALUES (value,...), (valueX,...) ...
```

A VALUES command with a single value set is equivalent to **SELECT value, ...** A VALUES command with multiple values sets is equivalent to a UNION ALL of simple SELECTs, for example **SELECT value, ... UNION ALL SELECT valueX, ...**

3.6.4. Update commands

Update commands report integer update counts. Update commands can report a maximum integer value of (2^31 -1). If you update a greater number of rows, the commands report the maximum integer value.

3.6.4.1. INSERT command

The INSERT command is used to add a record to a table.

Example syntax

```
INSERT INTO table (column,...) VALUES (value,...)
```

```
INSERT INTO table (column,...) query
```

3.6.4.2. UPDATE command

The UPDATE command is used to modify records in a table. The operation results in 1 or more records being updated, or in no records being updated if none match the criteria.

Example syntax

```
UPDATE table [[AS] alias] SET (column=value,...) [WHERE criteria]
```

3.6.4.3. DELETE command

The DELETE command is used to remove records from a table. The operation results in 1 or more records being deleted, or in no records being deleted if none match the criteria.

Example syntax

```
DELETE FROM table [[AS] alias] [WHERE criteria]
```

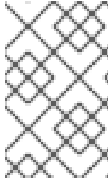
3.6.4.4. UPSERT (MERGE) command

The **UPSERT** (or **MERGE**) command is used to add or update records. The non-ANSI version of **UPSERT** that is implemented in Data Virtualization is a modified INSERT statement that requires that the target table has a primary key, and that the target columns cover the primary key. Before it performs an **INSERT**, the **UPSERT** operation checks whether a row exists, and if it does, **UPSERT** updates the current row rather than inserting a new one.

Example syntax

```
UPSERT INTO table [[AS] alias] (column,...) VALUES (value,...)
```

```
UPSERT INTO table (column,...) query
```



UPSERT PUSHDOWN

If an **UPSERT** statement is not pushed to the source, it is broken down into the respective INSERT and UPDATE operations. The target database system must support extended architecture (XA) to guarantee transaction atomicity.

3.6.4.5. EXECUTE command

The EXECUTE command is used to execute a procedure, such as a virtual procedure or a stored procedure. Procedures can have zero or more scalar input parameters. The return value from a procedure is a result set, or the set of inout/out/return scalars.

You can use the following short forms of the EXECUTE command:

- EXEC
- CALL

Example syntax

```
EXECUTE proc()
```

```
CALL proc(value, ...)
```

Named parameter syntax

```
EXECUTE proc(name1=>value1,name4=>param4, ...)
```

Syntax rules

- The default order of parameter specification is the same as how they are defined in the procedure definition.
- You can specify the parameters in any order by name. Parameters that have default values, or that are nullable in the metadata, can be omitted from the named parameter call, and will have the appropriate value passed at runtime.
- Positional parameters that have default values or that are nullable in the metadata, can be omitted from the end of the parameter list and will have the appropriate value passed at runtime.
- If the procedure does not return a result set, the values from the RETURN, OUT, and IN_OUT parameters are returned as a single row when used as an inline view query.
- A VARIADIC parameter may be repeated 0 or more times as the last positional argument.

3.6.4.6. Procedural relational command

Procedural relational commands use the syntax of a SELECT to emulate an EXEC. In a procedural relational command, a procedure group name is used in a FROM clause in place of a table. That procedure is executed in place of a normal table access if all of the necessary input values can be found in criteria against the procedure. Each combination of input values that is found in the criteria results in the execution of the procedure.

Example syntax

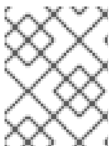
```
select * from proc
```

```
select output_param1, output_param2 from proc where input_param1 = 'x'
```

```
select output_param1, output_param2 from proc, table where input_param1 = table.col1 and
input_param2 = table.col2
```

Syntax rules

- The procedure as a table projects the same columns as an EXEC with the addition of the input parameters. For procedures that do not return a result set, IN_OUT columns are projected as two columns:
  - One to represents the output value.
  - One with the name {column name}_IN that represents the input of the parameter.
- Input values are passed via criteria. Values can be passed by **=**, **is null**, or as **in** predicates. Disjuncts are not allowed. It is also not possible to pass the value of a non-comparable column through an equality predicate.
- The procedure view automatically has an access pattern on its IN and IN_OUT parameters. The access pattern allows the procedure view to be planned correctly as a dependent join when necessary, or to fail when sufficient criteria cannot be found.
- Procedures that contain duplicate names between the parameters (IN, IN_OUT, OUT, RETURN) and the result set columns cannot be used in a procedural relational command.
- If there is already a table or view with the same name as the procedure, then it cannot be invoked via procedural relational syntax.
- Default values for IN or IN_OUT parameters are not used if there is no criteria present for a given input. Default values are only valid for named procedure syntax. For more information, see EXECUTE.



NOTE

The preceding issues do not apply when you use a nested table reference. For more information, see *Nested table reference* in FROM clause.

Multiple execution

The use of **in** or join criteria can result in a procedure being executed multiple times.

3.6.4.7. Anonymous procedure block

You can execute a procedure language block as a user command. This can be an advantage in situations in which a virtual procedure does not exist, but a set of processes can be carried out on the server side. For more information about the language for defining virtual procedures, see Procedure language.

Example syntax

```
begin insert into pm1.g1 (e1, e2) select ?, ?; select rowcount; end;
```

Syntax rules

- You can use **in** parameters with prepared/callable statement parameters, as shown in the preceding example, which uses **?** parameter.
- You cannot use **out** parameters in an anonymous procedure block. As a workaround, you can use session variables as needed.
- Anonymous procedure blocks do not return data as output parameters.
- A single result is returned if any of the statements returns a result set. All returnable result sets must have a matching number of columns and types. To indicate that a statement is not intended to provide a result set, use the WITHOUT RETURN clause.

3.6.5. Subqueries

A subquery is a SQL query embedded within another SQL query. The query containing the subquery is the outer query.

Subquery types:

- Scalar subquery - a subquery that returns only a single column with a single value. Scalar subqueries are a type of expression and can be used where single valued expressions are expected.
- Correlated subquery - a subquery that contains a column reference to from the outer query.
- Uncorrelated subquery - a subquery that contains no references to the outer sub-query.

Inline views

Subqueries in the FROM clause of the outer query (also known as "inline views") can return any number of rows and columns. This type of subquery must always be given an alias. An inline view is nearly identical to a traditional view. See also WITH Clause.

Example subquery in FROM clause (inline view)

```
SELECT a FROM (SELECT Y.b, Y.c FROM Y WHERE Y.d = '3') AS X WHERE a = X.c AND b = X.b
```

Subqueries can appear anywhere where an expression or criteria is expected.

You can use subqueries in quantified criteria, the **EXISTS** predicate, the **IN** predicate, and as Scalar subqueries.

Example subquery in WHERE using EXISTS

```
SELECT a FROM X WHERE EXISTS (SELECT 1 FROM Y WHERE c=X.a)
```

Example quantified comparison subqueries

```
SELECT a FROM X WHERE a >= ANY (SELECT b FROM Y WHERE c=3)
SELECT a FROM X WHERE a < SOME (SELECT b FROM Y WHERE c=4)
SELECT a FROM X WHERE a = ALL (SELECT b FROM Y WHERE c=2)
```

Example IN subquery

```
SELECT a FROM X WHERE a IN (SELECT b FROM Y WHERE c=3)
```

See also Subquery Optimization.

3.6.6. WITH clause

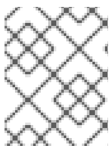
Data Virtualization provides access to common table expressions via the **WITH** clause. You can reference **WITH** clause items as tables in subsequent WITH clause items, and in the main query. You can think of the **WITH** clause as providing query-scoped temporary tables.

Usage

```
WITH name [(column, ...)] AS [/*+ no_inline|materialize */] (query expression) ...
```

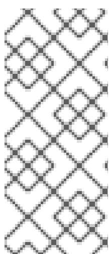
Syntax rules

- All of the projected column names must be unique. If they are not unique, then the column name list must be provided.
- If the columns of the WITH clause item are declared, then they must match the number of columns projected by the query expression.
- Each WITH clause item must have a unique name.
- The optional **no_inline** hint indicates to the optimizer that the query expression should not be substituted as an inline view where referenced. It is possible with no_inline for multiple evaluations of the common table as needed by source queries.
- The optional **materialize** hint requires that the common table be created as a temporary table in Data Virtualization. This forces a single evaluation of the common table.



NOTE

The WITH clause is also subject to optimization and its entries might not be processed if they are not needed in the subsequent query.



NOTE

Common tables are aggressively inlined to enhance the possibility of pushdown. If a common table is only referenced a single time in the main query, it is likely to be inlined. In some situations, such as when you use a common table to prevent n-many-processing of a non-pushdown, correlated subquery, you might need to include the **no_inline** or **materialize** hint.

Examples

```
WITH n (x) AS (select col from tbl) select x from n, n as n1
```

```
WITH n (x) AS /*+ no_inline */ (select col from tbl) select x from n, n as n1
```

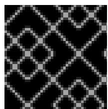
Recursive common table expressions

A recursive common table expression is a special form of a common table expression that is allowed to refer to itself to build the full common table result in a recursive or iterative fashion.

Usage

```
WITH name [(column, ...)] AS (anchor query expression UNION [ALL] recursive query expression) ...
```

The recursive query expression is allowed to refer to the common table by name. The anchor query expression is executed first during processing. Results are added to the common table and are referenced for the execution of the recursive query expression. The process is repeated against the new results until there are no more intermediate results.



IMPORTANT

Non-terminating, recursive common table expressions can lead to excessive processing.

By default, to prevent runaway processing of a recursive common table expression, processing is limited to 10000 iterations. Recursive common table expressions that are pushed down are not subject to this limit, but could be subject to other source-specific limits. You can modify the limit by setting the session variable **teiid.maxRecursion** to a larger integer value. After the limit is exceeded, an exception is thrown.

The following example fails, because the recursion limit is reached before processing completes.

```
SELECT teiid_session_set('teiid.maxRecursion', 25);
WITH n (x) AS (values('a') UNION select chr(ascii(x)+1) from n where x < 'z') select * from n
```

3.6.7. SELECT clause

SQL queries that start with the **SELECT** keyword and are often referred to as *SELECT statements*. YOu can use most of the standard SQL query constructs in Data Virtualization.

Usage

```
SELECT [DISTINCT|ALL] ((expression [[AS] name])|(group identifier.STAR))*|STAR ...
```

Syntax Rules

- Aliased expressions are only used as the output column names and in the ORDER BY clause. They cannot be used in other clauses of the query.
- DISTINCT may only be specified if the SELECT symbols are comparable.

3.6.8. FROM clause

The FROM clause specifies the target tables for SELECT, UPDATE, and DELETE statements.

Example Syntax:

- FROM table [[AS] alias]
- FROM table1 [INNER|LEFT OUTER|RIGHT OUTER|FULL OUTER] JOIN table2 ON join-criteria
- FROM table1 CROSS JOIN table2
- FROM (subquery) [AS] alias
- FROM TABLE(subquery) [AS] alias. For more information, see Nested tables
- FROM table1 JOIN /*+ MAKEDEP */ table2 ON join-criteria
- FROM table1 JOIN /*+ MAKENOTDEP */ table2 ON join-criteria
- FROM /*+ MAKEIND */ table1 JOIN table2 ON join-criteria
- FROM /*+ NO_UNNEST */ vw1 JOIN table2 ON join-criteria
- FROM table1 left outer join /*+ optional */ table2 ON join-criteria. For more information, see *Optional join* in Federated optimizations.
- FROM TEXTTABLE... For more information, see TEXTTABLE.
- FROM XMLTABLE... For more information, see XMLTABLE.
- FROM ARRAYTABLE... For more information, see ARRAYTABLE.
- FROM OBJECTTABLE... For more information, see OBJECTTABLE.
- FROM JSONTABLE... For more information, see JSONTABLE.
- FROM SELECT... For more information, see *Inline views* in Subqueries.

From clause hints

From clause hints are typically specified in a comment block preceding the affected clause. MAKEDEP and MAKENOTDEP may also appear after in non-comment form after the affected clause. If multiple hints apply to that clause, the hints should be placed in the same comment block.

Example hint

> FROM */*+ MAKEDEP PRESERVE */* (tbl1 inner join tbl2 inner join tbl3 on tbl2.col1 = tbl3.col1 on tbl1.col1 = tbl2.col1), tbl3 WHERE tbl1.col1 = tbl2.col1

Dependent join hints

MAKEIND, **MAKEDEP**, and **MAKENOTDEP** are hints that you can use to control dependent join behavior. Use them only in situations where the optimizer does not choose the most optimal plan based upon query structure, metadata, and costing information. The hints can appear in a comment that follows the **FROM** keyword. The hints can be specified against any **FROM** clause, not just a named table.

MAKEIND

Indicates that the clause should be the independent (feeder) side of a dependent join.

MAKEDEP

Indicates that the clause should be the dependent (filtered) side of a join.

MAKENOTDEP
: Prevents the clause from being the dependent (filtered) side of a join.

You can use the following optional **MAX** and **JOIN** arguments with **MAKEDEP** and **MAKEIND**:

MAKEDEP(JOIN)
: Indicates that the entire join should be pushed.

MAKEDEP(NO JOIN)
: Indicates that the entire join should not be pushed.

MAKEDEP(MAX:val)
: Indicates that the dependent join should only be performed if there are less than the maximum number of values from the independent side.

Other hints

NO_UNNEST can be specified against a subquery FROM clause or view to instruct the planner to not to merge the nested SQL in the surrounding query. This process is known as view flattening. This hint only applies to Data Virtualization planning and is not passed to source queries. NO_UNNEST can appear in a comment that follows the FROM keyword.

The PRESERVE hint can be used against an ANSI join tree to preserve the structure of the join, rather than allowing the Data Virtualization optimizer to reorder the join. This is similar in function to the Oracle ORDERED or MySQL STRAIGHT_JOIN hints.

Example PRESERVE hint

```
FROM /*+ PRESERVE */ (tbl1 inner join tbl2 inner join tbl3 on tbl2.col1 = tbl3.col1 on tbl1.col1 =
tbl2.col1)
```

3.6.8.1. Nested tables

Nested tables can appear in a **FROM** clause with the **TABLE** keyword. They are an alternative to using a view with normal join semantics. The columns projected from a command contained in a nested table can be used in join criteria, WHERE clauses, and other contexts where you can use FROM clause projected columns.

A nested table can have correlated references to preceding **FROM** clause column references as long as **INNER** and **LEFT OUTER** joins are used. This is especially useful in cases where then nested expression is a procedure or function call.

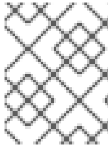
Valid nested table example

```
select * from t1, TABLE(call proc(t1.x)) t2
```

Invalid nested table example

The following nested table example is invalid, because **t1** appears after the nested table in the FROM clause:

```
select * from TABLE(call proc(t1.x)) t2, t1
```



MULTIPLE EXECUTION

Using a correlated nested table can result in multiple executions of the table expression – one for each correlated row.

3.6.8.2. XMLTABLE

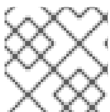
The XMLTABLE function uses XQuery to produce tabular output. The XMLTABLE function is implicitly a nested table and it can be used within FROM clauses. XMLTABLE is part of the SQL/XML 2006 specification.

Usage

```
XMLTABLE([<NSP>,] xquery-expression [<PASSING>] [COLUMNS <COLUMN>, ... ]) AS name
```

```
COLUMN := name (FOR ORDINALITY | (datatype [DEFAULT expression] [PATH string]))
```

For the definition of NSP - XMLNAMESPACES, see *XMLELEMENT* in XML functions. For the definition of PASSING, see *XMLQUERY* in XML functions.



NOTE

See also XQuery optimization.

Parameters

- The optional XMLNAMESPACES clause specifies the namepaces that you can use in the XQuery and COLUMN path expressions.
- The xquery-expression must be a valid XQuery. Each sequence item returned by the xquery is used to create a row of values as defined by the COLUMNS clause.
- If COLUMNS is not specified, that is equivalent to a COLUMNS clause that returns the entire item as an XML value, as in the following example:

  ```
  "COLUMNS OBJECT_VALUE XML PATH '.'"
  ```

- FOR ORDINALITY columns are typed as integers and return 1-based item numbers as their value.
- Each non-ordinality column specifies a type, and optionally specifies a PATH and a DEFAULT expression.
- If PATH is not specified, then the path is the same as the column name.

Syntax Rules

- You can specify only 1 FOR ORDINALITY column.
- Columns names must not contain duplicates.
- You can use binary large object (BLOB) datatypes, but there is built-in compatibility only for **xs:hexBinary** values. For xs:base64Binary, use a workaround of a PATH that uses the following explicit value constructor: **xs:base64Binary(<path>)**.

- The column expression must evaluate to a single value if a non-array type is expected.
- If an array type is specified, then an array is returned, unless there are no elements in the sequence, in which case a null value is returned.
- An empty element is not a valid null value, because its value is effectively an empty string. Use the **xsi:nil** attribute to specify a null value for an element.

XMLTABLE examples

Use of PASSING, returns 1 row [1]

```
select * from xmltable('/a' PASSING xmlparse(document '<a id="1"/>') COLUMNS id integer PATH
'@id') x
```

As a nested table

```
select x.* from t, xmltable('/x/y' PASSING t.doc COLUMNS first string, second FOR ORDINALITY) x
```

Invalid multi-value

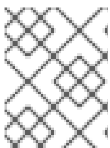
```
select * from xmltable('/a' PASSING xmlparse(document '<a><b id="1"/><b id="2"/></a>') COLUMNS
id integer PATH 'b/@id') x
```

Array multi-value

```
select * from xmltable('/a' PASSING xmlparse(document '<a><b id="1"/><b id="2"/></a>') COLUMNS
id integer[] PATH 'b/@id') x
```

Nil element

```
select * from xmltable('/a' PASSING xmlparse(document '<a
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"><b xsi:nil="true"/></a>') COLUMNS id
integer PATH 'b') x
```



NOTE

In the preceding example, an exception would be thrown if the **nil** attribute (**xsi:nil="true"**) were not specified, converting **b** to an integer value.

3.6.8.3. ARRAYTABLE

The ARRAYTABLE function processes an array input to produce tabular output. The function itself defines what columns it projects. The ARRAYTABLE function is implicitly a nested table and can be used within FROM clauses.

Usage

```
ARRAYTABLE([ROW|ROWS] expression COLUMNS <COLUMN>, ...) AS name
COLUMN := name datatype
```

Parameters

expression

The array to process, which should be a java.sql.Array or java array value.

ROW|ROWS

If ROW (the default) is specified, then only a single row is produced from the given array (typically a one dimensional array). If ROWS is specified, then multiple rows are produced. A multidimensional array is expected, and one row is produced for every non-null element of the outer array.

If the expression is null, no rows are produced.

Syntax rules

- Columns names cannot contain duplicates.

Array table examples

- As a nested table:

```
select x.* from (call source.invokeMDX('some query')) r, arraytable(r.tuple COLUMNS first string,
second bigdecimal) x
```

ARRAYTABLE is effectively a shortcut for using the **array_get** function in a nested table.

For example, the following ARRAYTABLE function:

```
ARRAYTABLE(val COLUMNS col1 string, col2 integer) AS X
```

is the same as the following statement which uses an **array_get** function:

```
TABLE(SELECT cast(array_get(val, 1) AS string) AS col1, cast(array_get(val, 2) AS integer) AS
col2) AS X
```

3.6.8.4. OBJECTTABLE

The OBJECTTABLE function processes an object input to produce tabular output. The function itself defines what columns it projects. The OBJECTTABLE function is implicitly a nested table and can be used within FROM clauses.

Usage

```
OBJECTTABLE([LANGUAGE lang] rowScript [PASSING val AS name ...] COLUMNS colName
colType colScript [DEFAULT defaultExpr] ...) AS id
```

Parameters

lang

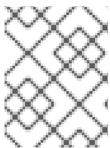
An optional string literal that is the case sensitive language name of the scripts to be processed. The script engine must be available via a JSR-223 ScriptEngineManager lookup.

If a LANGUAGE is not specified, the default 'teiid_script' is used. name:: An identifier that binds the **val** expression value into the script context. rowScript:: A string literal that specifies the script to create the

row values. For each non-null item that the Iterator produces, the columns are evaluated. colName/colType:: ID/data type of the column, which can optionally be defaulted with the DEFAULT clause expression **defaultExpr**. colScript:: A string literal that specifies the script that evaluates to the column value.

Syntax rules

- Columns names cannot contain duplicates.
- Data Virtualization places several special variables in the script execution context. The CommandContext is available as **teiid_context**. Additionally the **colScripts** may access **teiid_row** and **teiid_row_number**. **teiid_row** is the current row object produced by the row script. **teiid_row_number** is the current 1-based row number.
- **rowScript** is evaluated to an Iterator. If the results is already an Iterator, it is used directly. If the evaluation result is an Iteratable, Array, or Array type, then an Iterator is obtained. Any other Object will be treated as an Iterator of a single item. In all cases null row values are skipped.



NOTE

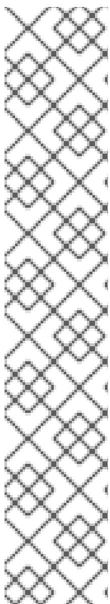
Although there are no restrictions on naming PASSING variables, it is best to choose names that you can reference as identifiers in the target language.

OBJECTTABLE examples

- Accessing special variables:

```
SELECT x.* FROM OBJECTTABLE('teiid_context' COLUMNS "user" string 'teiid_row.userName',
row_number integer 'teiid_row_number') AS x
```

The result would be a row with two columns containing the user name and 1 respectively.



NOTE

Languages other than teiid_script generally permit unrestricted access to Java functionality. As a result, by default, their use is restricted. You can override the restrictions by declaring allowable languages by name in the **allowed-languages** property. To use OBJECTTABLE, even from within view definitions that are not normally subject to permission checks, you must define the **allowed-languages** property. You must also set language access rights for user accounts to enable users to process OBJECTTABLE functions.

- For more information about about teiid_script, see the next section.
- For more information about enabling the use of languages other than teiid_script, see *allowed-languages* in Virtual database properties.
- For more information about setting user account permission, see *User query permissions* in Permissions.

teiid_script

teiid_script is a simple scripting expression language that allows access to passing and special variables, and to non-void 0-argument methods on objects and indexed values on arrays/lists. A teiid_script expression begins by referencing the passing or special variable. Then, any number of **.** accessors can be

chained to evaluate the expression to a different value. Methods may be accessed by their property names, for example, foo rather than getFoo. If the object includes both a **getFoo()** and **foo()** method, then the accessor **foo** references **foo ()**, and **getFoo** should be used to call the getter. An array or list index is accessed using a 1-based positive integral value, using the same **.** accessor syntax. The same logic as the system function **array_get** is used. That is, if the index is out of bounds, **null** is returned, rather than an exception.

teiid_script is effectively dynamically typed as typing is performed at runtime. If an accessor does not exist on the object, or if the method is not accessible, then an exception is raised. If any point in the accessor chain evaluates to a null value, then null will be returned.

teiid_script examples

- To get the VDB description string:

```
teiid_context.session.vdb.description
```

- To get the first character of the VDB description string:

```
teiid_context.session.vdb.description.toCharArray.1
```

3.6.8.5. TEXTTABLE

The TEXTTABLE function processes character input to produce tabular output. It provides both fixed and delimited file format parsing. The function itself defines what columns it projects. The TEXTTABLE function is implicitly a nested table and can be used within FROM clauses.

Usage

```
TEXTTABLE(expression [SELECTOR string] COLUMNS <COLUMN>, ... [NO ROW DELIMITER |
ROW DELIMITER char] [DELIMITER char] [(QUOTE|ESCAPE) char] [HEADER [integer]] [SKIP
integer] [NO TRIM]) AS name
```

Where <COLUMN>

```
COLUMN := name (FOR ORDINALITY | ([HEADER string] datatype [WIDTH integer [NO TRIM]]
[SELECTOR string integer]))
```

Parameters

expression

The text content to process, which should be convertible to a character large object (CLOB).

SELECTOR

Used with files containing multiple types of rows (example: order header, detail, summary). A TEXTTABLE SELECTOR specifies which lines to include in the output. Matching lines must begin with the selector string. The selector in column delimited files must be followed by the column delimiter.
If a TEXTTABLE SELECTOR is specified, a SELECTOR may also be specified for column values. A column SELECTOR argument will select the nearest preceding text line with the given SELECTOR prefix, and select the value at the given 1-based integer position (which includes the selector itself). If no such text line or position with a given line exists, a null value will be produced. A column SELECTOR is not valid with fixed width parsing.

NO ROW DELIMITER

Specifies that fixed parsing should not assume the presence of newline row delimiters.

ROW DELIMITER

Sets the row delimiter / newline to an alternate character. Defaults to the new-line character - with built-in handling for treating carriage return newline as a single character. If ROW DELIMITER is specified, carriage return is given no special treatment.

DELIMITER

Sets the field delimiter character to use. Defaults to **,**.

QUOTE

Sets the quote, or qualifier, character used to wrap field values. Defaults to **"**.

ESCAPE

Sets the escape character to use if no quoting character is in use. This is used in situations where the delimiter or new line characters are escaped with a preceding character, e.g. \.

HEADER

Specifies the text line number (counting every new line) on which the column names occur. If the HEADER option for a column is specified, then that will be used as the expected header name. All lines prior to the header will be skipped. If HEADER is specified, then the header line will be used to determine the TEXTTABLE column position by case-insensitive name matching. This is especially useful in situations where only a subset of the columns are needed. If the HEADER value is not specified, it defaults to 1. If HEADER is not specified, then columns are expected to match positionally with the text contents.

SKIP

Specifies the number of text lines (counting every new line) to skip before parsing the contents. HEADER can be specified with SKIP.

FOR ORDINALITY

Column that is typed as integer and returns a 1-based item number as its value.

WIDTH

Indicates the fixed-width length of a column in characters, not bytes. With the default ROW DELIMITER, a CR NL sequence counts as a single character.

NO TRIM

When specified on a TEXTTABLE, it affects all column and header values. When NO TRIM is specified on a column, the fixed or unqualified text value is not trimmed of leading and trailing white space.

Syntax Rules

- If width is specified for one column it must be specified for all columns and be a non-negative integer.
- If width is specified, then fixed width parsing is used, and ESCAPE, QUOTE, column SELECTOR, nor HEADER should not be specified.
- If width is not specified, then NO ROW DELIMITER cannot be used.
- Columns names must not contain duplicates.
- The characters specified for QUOTE, DELIMITER, and ROW DELIMITER must all be different.

TEXTTABLE examples

- Use of the HEADER parameter, returns 1 row ['b']:

```
SELECT * FROM TEXTTABLE(UNESCAPE('col1,col2,col3\na,b,c') COLUMNS col2 string HEADER)
x
```

- Use of fixed width, returns 2 rows ['a', 'b', 'c'], ['d', 'e', 'f']:

```
SELECT * FROM TEXTTABLE(UNESCAPE('abc\ndef') COLUMNS col1 string width 1, col2 string
width 1, col3 string width 1) x
```

- Use of fixed width without a row delimiter, returns 3 rows ['a'], ['b'], ['c']:

```
SELECT * FROM TEXTTABLE('abc' COLUMNS col1 string width 1 NO ROW DELIMITER) x
```

- Use of ESCAPE parameter, returns 1 row ['a,', 'b']:

```
SELECT * FROM TEXTTABLE('a:,,b' COLUMNS col1 string, col2 string ESCAPE ':') x
```

- As a nested table:

```
SELECT x.* FROM t, TEXTTABLE(t.clobcolumn COLUMNS first string, second date SKIP 1) x
```

- Use of SELECTORs, returns 2 rows ['c', 'd', 'b'], ['c', 'f', 'b']:

```
SELECT * FROM TEXTTABLE('a,b\nc,d\nc,f' SELECTOR 'c' COLUMNS col1 string, col2 string col3
string SELECTOR 'a' 2) x
```

3.6.8.6. JSONTABLE

The JSONTABLE function uses JsonPath to produce tabular output. The JSONTABLE function is implicitly a nested table and can be used within FROM clauses.

Usage

```
JSONTABLE(value, path [, nullLeafOnMissing] COLUMNS <COLUMN>, ... ) AS name
```

```
COLUMN := name (FOR ORDINALITY | (datatype [PATH string]))
```

See also JsonPath

Parameters

value

A clob containing a valid JSON document.

nullLeafOnMissing

If false (the default), then a path that evaluates to a leaf that is missing will throw an exception. If nullLeafOnMissing is true, then a null value will be returned.

PATH

String should be a valid JsonPath. If an array value is returned, then each non-null element will be used to generate a row. Otherwise a single non-null item will be used to create a single row.

FOR ORDINALITY

Column typed as integer. Returns a 1-based item number as its value.

- Each non-ordinality column specifies a type and optionally a PATH.
- If PATH is not specified, then the path will be generated from the column name: @['name'], which will look for an object key value matching name. If PATH is specified, it must begin with @, which means that the path will be processed relative the the current row context item.

Syntax Rules

- Columns names must not contain duplicates.
- You cannot use array types with the JSONTABLE function.

JSONTABLE examples

Use of passing, returns 1 row [1]:

```
select * from jsontable('{"a": {"id":1}}}', '$.a' COLUMNS id integer) x
```

As a nested table:

```
select x.* from t, jsontable(t.doc, '$.x.y' COLUMNS first string, second FOR ORDINALITY) x
```

With more complicated paths:

```
select x.* from jsontable('[{"firstName": "John", "lastName": "Wayne", "children": []}, {"firstName":
"John", "lastName": "Adams", "children":["Sue","Bob"]}]', '$.*' COLUMNS familyName string path
'@.lastName', children integer path '@.children.length()' ) x
```

Differences with XMLTABLE

Processing of JSON to tabular results was previously recommended through the use of XMLTABLE with JSONTOXML. For most tasks, JSONTABLE provides a simpler syntax. However, there are some differences to consider:

- JSONTABLE parses the JSON completely, the processes it. XMLTABLE uses streaming processing to reduce the memory overhead.
- JsonPath is not as powerful as XQuery. There are a lot of functions and operations available in XQuery/XPath that are not available in JsonPath.
- JsonPath does not allow for parent references in the column paths. There is no ability to reference the root or any part of the parent hierarchy (.. in XPath).

3.6.9. WHERE clause

The WHERE clause defines the criteria to limit the records affected by SELECT, UPDATE, and DELETE statements.

The general form of the WHERE is:

- WHERE Criteria

3.6.10. GROUP BY clause

The GROUP BY clause denotes that rows should be grouped according to the specified expression values. One row is returned for each group, after optionally filtering those aggregate rows based on a HAVING clause.

The general form of the GROUP BY is:

```
GROUP BY expression [,expression]*
```

```
GROUP BY ROLLUP(expression [,expression]*)
```

Syntax Rules

- Column references in the group by cannot be made to alias names in the SELECT clause.
- Expressions used in the group by must appear in the select clause.
- Column references and expressions in the SELECT/HAVING/ORDER BY clauses that are not used in the group by clause must appear in aggregate functions.
- If an aggregate function is used in the SELECT clause and no GROUP BY is specified, an implicit GROUP BY will be performed with the entire result set as a single group. In this case, every column in the SELECT must be an aggregate function as no other column value will be fixed across the entire group.
- The GROUP BY columns must be of a comparable type.

Rollups

Just like normal grouping, ROLLUP processing logically occurs before the HAVING clause is processed. A ROLLUP of expressions will produce the same output as a regular grouping with the addition of aggregate values computed at higher aggregation levels. For N expressions in the ROLLUP, aggregates will be provided over (), (expr1), (expr1, expr2), etc. up to (expr1, … exprN-1), with the other grouping expressions in the output as null values. The following example uses a normal aggregation query:

```
SELECT country, city, sum(amount) from sales group by country, city
```

The query returns the following data:

Table 3.1. Data returned by a normal aggregation query

country	city	sum(amount)
US	St. Louis	10000
US	Raleigh	150000
US	Denver	20000
UK	Birmingham	50000
UK	London	75000

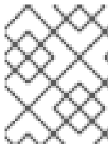
In contrast, the following example uses a rollup query:

Data returned from a rollup query

```
SELECT country, city, sum(amount) from sales group by rollup(country, city)
```

would return:

country	city	sum(amount)
US	St. Louis	10000
US	Raleigh	150000
US	Denver	20000
US	**<null>**	180000
UK	Birmingham	50000
UK	London	75000
UK	**<null>**	125000
<null>	**<null>**	305000



NOTE

Not all sources are compatible with ROLLUPs, and compared to normal aggregate processing, some optimizations might be inhibited by the use of a ROLLUP.

The use of ROLLUPs in Data Virtualization is currently limited in comparison to the SQL specification.

3.6.11. HAVING Clause

The **HAVING** clause operates exactly as a **WHERE** clause, although it operates on the output of a **GROUP BY**. You can use the same syntax with the **HAVING** clause as with the **WHERE** clause.

Syntax Rules

- Expressions used in the GROUP BY clause must contain either an aggregate function (**COUNT**, **AVG**, **SUM**, **MIN**, **MAX**), or be one of the grouping expressions.

3.6.12. ORDER BY clause

The ORDER BY clause specifies how records are sorted. The options are ASC (ascending) or DESC (descending).

Usage

```
ORDER BY expression [ASC|DESC] [NULLS (FIRST|LAST)], ...
```

Syntax rules

- Sort columns can be specified positionally by a 1-based positional integer, by SELECT clause alias name, by SELECT clause expression, or by an unrelated expression.
- Column references can appear in the SELECT clause as the expression for an aliased column, or can reference columns from tables in the FROM clause. If the column reference is not in the SELECT clause, the query cannot be a set operation, specify SELECT DISTINCT, or contain a GROUP BY clause.
- Unrelated expressions, expressions not appearing as an aliased expression in the select clause, are allowed in the ORDER BY clause of a non-set QUERY. The columns referenced in the expression must come from the from clause table references. The column references cannot be to alias names or positional.
- The ORDER BY columns must be of a comparable type.
- If an ORDER BY is used in an inline view or view definition without a LIMIT clause, it is removed by the Data Virtualization optimizer.
- If NULLS FIRST/LAST is specified, then nulls are guaranteed to be sorted either first or last. If the null ordering is not specified, then results will typically be sorted with nulls as low values, which is the default internal sorting behavior for Data Virtualization. However, not all sources return results with nulls sorted as low values by default, and Data Virtualization might return results with different null orderings.

WARNING

The use of positional ordering is no longer supported by the ANSI SQL standard and is a deprecated feature in Data Virtualization. It is best to use alias names in the ORDER BY clause.

3.6.13. LIMIT clause

The LIMIT clause specifies a limit on the number of records returned from the SELECT command. YOu can specify an optional offset (the number of rows to skip). The LIMIT clause can also be specified using the SQL 2008 OFFSET/FETCH FIRST clauses. If an ORDER BY is also specified, it will be applied before the OFFSET/LIMIT are applied. If an ORDER BY is not specified there is generally no guarantee what subset of rows will be returned.

Usage

```
LIMIT [offset,] limit
```

```
LIMIT limit OFFSET offset
```

```
[OFFSET offset ROW|ROWS] [FETCH FIRST|NEXT [limit] ROW|ROWS ONLY]
```

Syntax rules

- The LIMIT/OFFSET expressions must be a non-negative integer or a parameter reference (**?**). An offset of **0** is ignored. A limit of **0** returns no rows.
- The terms FIRST/NEXT are interchangeable as well as ROW/ROWS.
- The LIMIT clause can take an optional preceding NON_STRICT hint to indicate that push operations should not be inhibited, even if the results will not be consistent with the logical application of the limit. The hint is only needed on unordered limits, for example, **"SELECT * FROM VW /*+ NON_STRICT */ LIMIT 2"**.

LIMIT clause examples

- **LIMIT 100** returns the first 100 records (rows 1-100).
- **LIMIT 500, 100** skips 500 records and returns the next 100 records(rows 501-600).
- **OFFSET 500 ROWS** skips 500 records.
- **OFFSET 500 ROWS FETCH NEXT 100 ROWS ONLY** skips 500 records and returns the next 100 records (rows 501-600).
- **FETCH FIRST ROW ONLY** returns only the first record.

3.6.14. INTO clause

WARNING

Usage of the INTO Clause for inserting into a table has been been deprecated. An INSERT with a query command should be used instead. For information about using INSERT, see INSERT command.

When the into clause is specified with a SELECT, the results of the query are inserted into the specified table. This is often used to insert records into a temporary table. The INTO clause immediately precedes the FROM clause.

Usage

```
INTO table FROM ...
```

Syntax rules

- The **INTO** clause is logically applied last in processing, after the **ORDER BY** and **LIMIT** clauses.
- Data Virtualization's support for **SELECT INTO** is similar to Microsoft SQL Server. The target of the **INTO** clause is a table where the result of the **SELECT** command will be inserted. For example, the following statement:

```
SELECT col1, col2 INTO targetTable FROM sourceTable
```

- inserts **col1** and **col2** from the **sourceTable** into the **targetTable**.

- You cannot combine SELECT INTO with a UNION query.
  That is, you cannot select the results from a **sourceTable UNION** query for insertion into a **targetTable**.

3.6.15. OPTION clause

The OPTION keyword denotes options that a user can pass in with a command. These options are specific to Data Virtualization and are not covered by any SQL specification.

Usage

```
OPTION option (, option)*
```

Supported options

MAKEDEP table (,table)*

Specifies source tables that should be made dependent in the join.

MAKEIND table (,table)*

Specifies source tables that should be made independent in the join.

MAKENOTDEP table (,table)*

Prevents a dependent join from being used.

NOCACHE [table (,table)*]

Prevents cache from being used for all tables or for the given tables.

Examples

```
OPTION MAKEDEP table1
```

```
OPTION NOCACHE
```

All tables specified in the OPTION clause should be fully qualified. However, the table name can match either the fully qualified name or an alias name.

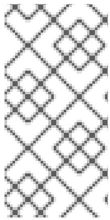
The MAKEDEP and MAKEIND hints can take optional arguments to control the dependent join. The extended hint form is:

```
MAKEDEP tbl([max:val] [[no] join])
```

- **tbl(JOIN)** means that the entire join should be pushed.
- **tbl(NO JOIN)** means that the entire join should not be pushed.
- **tbl(MAX:val)** means that the dependent join should only be performed if there are less than the maximum number of values from the independent side.

TIP

Data Virtualization does not accept PLANONLY, DEBUG, and SHOWPLAN arguments in the OPTION clause. For information about how to perform the functions formerly provided by these options, see the Client Developer's Guide.



NOTE

MAKEDEP and MAKENOTDEP hints can take table names in the form of **@view1.view2...table**. For example, with an inline view **"SELECT * FROM (SELECT * FROM tbl1, tbl2 WHERE tbl1.c1 = tbl2.c2) AS v1 OPTION MAKEDEP @v1.tbl1"** the hint will now be understood as applying under the v1 view.

3.7. DDL COMMANDS

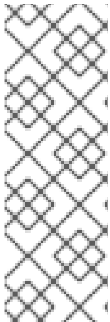
Data Virtualization is compatible with a subset of the DDL commands for creating or dropping temporary tables and manipulating procedure and view definitions at runtime. It is not currently possible to arbitrarily drop or create non-temporary metadata entries. For information about the DDL statements that you can use to define schemas in a virtual database, see DDL metadata.

3.7.1. Temporary Tables

You can create and use temporary (*temp*) tables in Data Virtualization. Temporary tables are created dynamically, but are treated as any other physical table.

3.7.1.1. Local temporary tables

Local temporary tables can be defined implicitly by referencing them in a INSERT statement or explicitly with a CREATE TABLE statement. Implicitly created temp tables must have a name that starts with **#**.



NOTE

Data Virtualization interprets *local* to mean that a temporary table is scoped to the session or block of the virtual procedure that creates it. This interpretation differs from the SQL specification and from the interpretation that other database vendors implement. After exiting a block or at the termination of a session, the table is dropped. Session tables and other temporary tables that a calling procedures creates are not visible to called procedures. If a temporary table of the same name is created in a called procedure, then a new instance is created.

Creation syntax

You can create local temporary tables explicitly or implicitly.

Explicit creation syntax

Local temporary tables can be defined explicitly with a CREATE TABLE statement, as in the following example:name: value

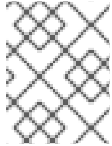
```
CREATE LOCAL TEMPORARY TABLE name (column type [NOT NULL], ... [PRIMARY KEY
(column, ...)]) [ON COMMIT PRESERVE ROWS]
```

- Use the SERIAL data type to specify a NOT NULL and auto-incrementing INTEGER column. The starting value of a SERIAL column is 1.

Implicit creation syntax

Local temporary tables can be defined implicitly by referencing them in an INSERT statement.

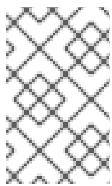
```
INSERT INTO #name (column, ...) VALUES (value, ...)
INSERT INTO #name [(column, ...)] select c1, c2 from t
```



NOTE

If **#name** does not exist, it is defined using the given column names and types from the value expressions.

```
INSERT INTO #name (column, ...) VALUES (value, ...)
INSERT INTO #name [(column, ...)] select c1, c2 from t
```



NOTE

If **#name** does not exist, it is defined using the target column names, and the types from the query-derived columns. If target columns are not supplied, the column names will match the derived column names from the query.

Drop syntax

```
DROP TABLE name
```

+ In the following example, a series of statements loads a temporary table with data from 2 sources, manually inserts a record, and then uses the temporary table in a SELECT query.

Example: Local temporary tables

```
CREATE LOCAL TEMPORARY TABLE TEMP (a integer, b integer, c integer);
SELECT * INTO temp FROM Src1;
SELECT * INTO temp FROM Src2;
INSERT INTO temp VALUES (1,2,3);
SELECT a,b,c FROM Src3, temp WHERE Src3.a = temp.b;
```

For more information about using local temporary tables, see Virtual procedures.

3.7.1.2. Global temporary tables

Global temporary tables are created from the metadata that you supply to Data Virtualization at deployment time. Unlike local temporary tables, you cannot create global temporary tables at runtime. Your global temporary tables share a common definition through a schema entry. However, a new instance of the temporary table is created in each session. The table is then dropped when the session ends. There is no explicit drop support. A common use for a global temporary table is to pass results into and out of procedures.

Creation syntax

```
CREATE GLOBAL TEMPORARY TABLE name (column type [NOT NULL], ... [PRIMARY KEY
(column, ...)]) OPTIONS (UPDATABLE 'true')
```

If you use the SERIAL data type, then each session's instance of the global temporary table will have its own sequence.

You must explicitly specify UPDATABLE if you want to update the temporary table.

For information about syntax options, see the **CREATE TABLE** DDL statements in DDL metadata for schema objects.

3.7.1.3. Common features of global and local temporary tables

Global and local temporary tables share some common features.

Primary key usage

- All key columns must be comparable.
- If you use a primary key, it creates a clustered index that enables search improvements for SQL comparison operators, and the IN, LIKE, and ORDER BY operators.
- You can use **Null** as a primary key value, but there must be only one row that has an all-null key.

Transactions

- There is a **READ_UNCOMMITED** transaction isolation level. There are no locking mechanisms available to enable higher isolation levels, and the result of a rollback may be inconsistent across multiple transactions. If concurrent transactions are not associated with the same local temporary table or session, then the transaction isolation level is effectively serializable. If you want full consistency with local temporary tables, then only use a connection with one transaction at a time. This mode of operation is ensured by connection pooling that tracks connections by transaction.

Limitations

- With the **CREATE TABLE** syntax, you can specify only basic table definition (column name, type, and nullable information), and an optional primary key. For global temporary tables, additional metadata in the CREATE statement is effectively ignored when creating the temporary table instance. However, the metadata might still be used by planning similar to any other table entry.
- You can use **ON COMMIT PRESERVE ROWS**. You cannot use other **ON COMMIT** actions.
- The cannot use "drop behavior" options in the DROP statement.
- Temporary tables are not fail-over safe.
- Non-inlined LOB values (XML, CLOB, BLOB, JSON, geometry) are tracked by reference rather than by value in a temporary table. If you insert LOB values from external sources in your temporary table, they might become unreadable when the associated statement or connection is closed.

3.7.1.4. Foreign temporary tables

Unlike a local or global temporary table, a foreign temporary table is a reference to an actual source table that is created at runtime, rather than during the metadata load.

A foreign temporary table requires explicit creation syntax:

```
CREATE FOREIGN TEMPORARY TABLE name ... ON schema
```

Where the table creation body syntax is the same as a standard CREATE FOREIGN TABLE DDL statement. For more information, see DDL metadata. In general, usage of DDL OPTION clauses might be required to properly access the source table, including setting the name in the source, updatability, native types, and so forth.

The schema name must specify an existing schema/model in the VDB. The table will be accessed as if it is on that source. However within Data Virtualization the temporary table will still be scoped the same as a non-foreign temporary table. This means that the foreign temporary table will not belong to a Data Virtualization schema, and will be scoped to the session or procedure block where it is created.

The DROP syntax for a foreign temporary table is the same as for a non-foreign temporary table.

Neither a CREATE nor a corresponding DROP of a foreign temporary table issues a pushdown command. Rather, this mechanism exposes a source table for use within Data Virtualization on a temporary basis.

There are two usage scenarios for a FOREIGN TEMPORARY TABLE. The first is to dynamically access additional tables on the source. The other is to replace the usage of a Data Virtualization local temporary table for performance reasons. The usage pattern for the latter case would look like:

```
//- create the source table
source.native("CREATE GLOBAL TEMPORARY TABLE name IF NOT EXISTS ... ON COMMIT
DELETE ROWS");
//- bring the table into Data Virtualization
CREATE FOREIGN TEMPORARY TABLE name ... OPTIONS (UPDATABLE true)
//- use the table
...
//- forget the table
DROP TABLE name
```

Note the usage of the native procedure to pass source-specific CREATE DDL to the source. Data Virtualization does not currently attempt to pushdown a source creation of a temporary table based on the CREATE statement. Some other mechanism, such as the native procedure shown above, must be used to first create the table. Also note the table is explicitly marked as updatable, since DDL defined tables are not updatable by default.

The source’s handling of temporary tables must also be understood to make this work as intended. Sources that use the same GLOBAL table definition for all sessions while scoping the data to be session-specific (such as Oracle) or sources that use session-scoped temporary tables (such as PostgreSQL) will work if accessed under a transaction. A transaction is necessary for the following reasons:

- The source on commit behavior (most likely DELETE ROWS or DROP) will ensure clean-up. Keep in mind that a Data Virtualization drop does not issue a source command and is not guaranteed to occur (in some exception cases, loss of database connectivity, hard shutdown, and so forth).

- The source pool when using track connections by transaction will ensure that multiple uses of that source by Data Virtualization will use the same connection/session and thus the same temporary table and data.

TIP

You cannot use the **ON COMMIT** clause with Data Virtualization. As a result, for local temporary tables, the **ON COMMIT** behavior for source tables is likely to be different from the default **PRESERVE ROWS**.

3.7.2. Alter view

Usage

```
ALTER VIEW name AS queryExpression
```

Syntax rules

- The alter query expression can be prefixed with a cache hint for materialized view definitions. The hint takes effect the next time that the materialized view table loads.

3.7.3. Alter procedure

Usage

```
ALTER PROCEDURE name AS block
```

Syntax rules

- The ALTER block should not include **CREATE VIRTUAL PROCEDURE**.
- You can prefix the ALTER block with a cache hint for cached procedures.

3.7.4. Alter trigger

Usage

```
ALTER TRIGGER ON name INSTEAD OF INSERT|UPDATE|DELETE (AS FOR EACH ROW block)
| (ENABLED|DISABLED)
```

Syntax rules

- The target **name** must be an updatable view.
- Triggers are not true schema objects. They are scoped only to their view and have no name.
- Update procedures must already exist for the given trigger event. For more information, see Triggers.

3.8. PROCEDURES

You can use a procedure language in Data Virtualization to call foreign procedures and define virtual procedures and triggers.

3.8.1. Procedure language

You can use a procedural language in Data Virtualization to define virtual procedures. These are similar to stored procedures in relational database management systems. You can use this language to define the transformation logic for decomposing INSERT, UPDATE, and DELETE commands against views. These are known as update procedures. For more information, see Virtual procedures and update procedures (Triggers).

3.8.1.1. Command statement

A command statement executes a DML command, DDL command, or dynamic SQL against one or more data sources. For more information, see DML commands and DDL commands.

Usage

```
command [(WITH|WITHOUT) RETURN];
```

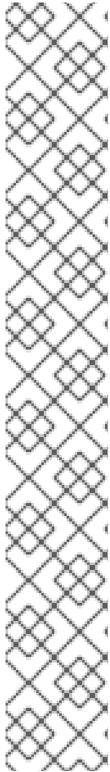
Example command statements

```
SELECT * FROM MySchema.MyTable WHERE ColA > 100 WITHOUT RETURN;
INSERT INTO MySchema.MyTable (ColA,ColB) VALUES (50, 'hi');
```

Syntax rules

- EXECUTE command statements may access IN/OUT, OUT, and RETURN parameters. To access the return value the statement will have the form **var = EXEC proc…**. To access OUT or IN/OUT values named parameter syntax must be used. For example, **EXEC proc(in_param⇒'1', out_param⇒var)** will assign the value of the out parameter to the variable var. It is expected that the datatype of a parameter is implicitly convertible to the data type of the variable. For more information about EXECUTE command statements, see EXECUTE command.

- The RETURN clause determines if the result of the command is returnable from the procedure. WITH RETURN is the default. If the command does not return a result set, or the procedure does not return a result set, the RETURN clause is ignored. If WITH RETURN is specified, the result set of the command must match the expected result set of the procedure. Only the last successfully executed statement executed WITH RETURN will be returned as the procedure result set. If there are no returnable result sets and the procedure declares that a result set will be returned, then an empty result set is returned.



NOTE

The INTO clause is used only for inserting into a table. `SELECT … INTO table … **is functionally equivalent to `INSERT INTO table SELECT …** If you need to assign variables, you can use one of the following methods:

Use an assignment statement with a scalar subquery

```
DECLARE string var = (SELECT col ...);
```

Use a temporary table

```
INSERT INTO #temp SELECT col1, col2 ...;
DECLARE string VARIABLES.RESULT = (SELECT x FROM #temp);
```

Use an array

```
DECLARE string[] var = (SELECT (col1, col2) ...);
DECLARE string col1val = var[1];
```

3.8.1.2. Dynamic SQL command

Dynamic SQL allows for the execution of an arbitrary SQL command in a virtual procedure. Dynamic SQL is useful in situations where the exact command form is not known prior to execution.

Usage

```
EXECUTE IMMEDIATE <sql expression> AS <variable> <type> [, <variable> <type>]* [INTO
<variable>] [USING <variable>=<expression> [,<variable>=<expression>]*] [UPDATE <literal>]
```

Syntax rules

- The SQL expression must be a CLOB or string value of less than 262144 characters.
- The **AS** clause is used to define the projected symbols names and types returned by the executed SQL string. The **AS** clause symbols will be matched positionally with the symbols returned by the executed SQL string. Non-convertible types or too few columns returned by the executed SQL string will result in an error.
- The **INTO** clause will project the dynamic SQL into the specified temp table. With the **INTO** clause specified, the dynamic command will actually execute a statement that behaves like an INSERT with a QUERY EXPRESSION. If the dynamic SQL command creates a temporary table with the **INTO** clause, then the **AS** clause is required to define the table's metadata.
- The **USING** clause allows the dynamic SQL string to contain variable references that are bound at runtime to specified values. This allows for some independence of the SQL string from the surrounding procedure variable names and input names. In the dynamic command **USING** clause, each variable is specified by short name only. However, in the dynamic SQL the **USING** variable must be fully qualified to **DVAR**. The **USING** clause is only for values that will be used in the dynamic SQL as valid expressions. It is not possible to use the **USING** clause to replace table names, keywords, and so forth. This makes using symbols equivalent in power to normal

bind (?) expressions in prepared statements. The **USING** clause helps reduce the amount of string manipulation needed. If a reference is made to a USING symbol in the SQL string that is not bound to a value in the **USING** clause, an exception will occur.

- The **UPDATE** clause is used to specify the updating model count. Accepted values are (0,1,*). 0 is the default value if the clause is not specified. For more information, see Updating model count.

Example: Dynamic SQL

```
...
/* Typically complex criteria would be formed based upon inputs to the procedure.
 In this simple example the criteria is references the using clause to isolate
 the SQL string from referencing a value from the procedure directly */

DECLARE string criteria = 'Customer.Accounts.Last = DVARS.LastName';

/* Now we create the desired SQL string */
DECLARE string sql_string = 'SELECT ID, First || " " || Last AS Name, Birthdate FROM
Customer.Accounts WHERE ' || criteria;

/* The execution of the SQL string will create the #temp table with the columns (ID, Name, Birthdate).
  Note that we also have the USING clause to bind a value to LastName, which is referenced in the
criteria. */
EXECUTE IMMEDIATE sql_string AS ID integer, Name string, Birthdate date INTO #temp USING
LastName='some name';

/* The temp table can now be used with the values from the Dynamic SQL */
loop on (SELCT ID from #temp) as myCursor
...
```

Here is an example showing a more complex approach to building criteria for the dynamic SQL string. In short, the virtual procedure **AccountAccess.GetAccounts** has the inputs **ID**, **LastName**, and **bday**. If a value is specified for **ID** it will be the only value used in the dynamic SQL criteria. Otherwise, if a value is specified for **LastName** the procedure will detect if the value is a search string. If **bday** is specified in addition to **LastName**, it will be used to form compound criteria with **LastName**.

Example: Dynamic SQL with USING clause and dynamically built criteria string

```
...
DECLARE string crit = null;

IF (AccountAccess.GetAccounts.ID IS NOT NULL)
 crit = '(Customer.Accounts.ID = DVARS.ID)';
ELSE IF (AccountAccess.GetAccounts.LastName IS NOT NULL)
BEGIN
 IF (AccountAccess.GetAccounts.LastName == '%')
   ERROR "Last name cannot be %";
 ELSE IF (LOCATE('%', AccountAccess.GetAccounts.LastName) < 0)
   crit = '(Customer.Accounts.Last = DVARS.LastName)';
 ELSE
   crit = '(Customer.Accounts.Last LIKE DVARS.LastName)';
 IF (AccountAccess.GetAccounts.bday IS NOT NULL)
   crit = '(' || crit || ' and (Customer.Accounts.Birthdate = DVARS.BirthDay))';
END
```

```
ELSE
 ERROR "ID or LastName must be specified.";

EXECUTE IMMEDIATE 'SELECT ID, First || " " || Last AS Name, Birthdate FROM
Customer.Accounts WHERE ' || crit USING ID=AccountAccess.GetAccounts.ID,
LastName=AccountAccess.GetAccounts.LastName, BirthDay=AccountAccess.GetAccounts.Bday;
...
```

Dynamic SQL limitations and workarounds

The use of the dynamic SQL command results in an assignment statement that requires the use of a temporary table.

Example assignment

```
EXECUTE IMMEDIATE <expression> AS x string INTO #temp;
DECLARE string VARIABLES.RESULT = (SELECT x FROM #temp);
```

The construction of appropriate criteria will be cumbersome if parts of the criteria are not present. For example if **criteria** were already NULL, then the following example results in **criteria** remaining NULL.

Example: Dangerous NULL handling

```
...
criteria = '(' || criteria || ' and (Customer.Accounts.Birthdate = DVARS.BirthDay))';
```

It is best to ensure that the criteria is not NULL prior its usage. If this is not possible, a you can specify a default, as shown in the following example.

Example: NULL handling

```
...
criteria = '(' || nvl(criteria, '(1 = 1)') || ' and (Customer.Accounts.Birthdate = DVARS.BirthDay))';
```

If the dynamic SQL is an **UPDATE**, **DELETE**, or **INSERT** command, the rowcount of the statement can be obtained from the rowcount variable.

Example: AS and INTO clauses

```
/* Execute an update */
EXECUTE IMMEDIATE <expression>;
```

3.8.1.3. Declaration statement

A declaration statement declares a variable and its type. After you declare a variable, you can use it in that block within the procedure and any sub-blocks. A variable is initialized to null by default, but can also be assigned the value of an expression as part of the declaration statement.

Usage

```
DECLARE <type> [VARIABLES.]<name> [= <expression>];
```

Example syntax

```
declare integer x;
declare string VARIABLES.myvar = 'value';
```

Syntax rules

- You cannot redeclare a variable with a duplicate name in a sub-block.
- The VARIABLES group is always implied even if it is not specified.
- The assignment value follows the same rules as for an Assignment statement.
- In addition to the standard types, you may specify EXCEPTION if declaring an exception variable.

3.8.1.4. Assignment statement

An assignment statement assigns a value to a variable by evaluating an expression.

Usage

```
<variable reference> = <expression>;
```

Example syntax

```
myString = 'Thank you';
VARIABLES.x = (SELECT Column1 FROM MySchema.MyTable);
```

Valid variables for assignment include any in-scope variable that has been declared with a declaration statement, or the procedure **in_out** and **out** parameters. **In_out** and **out** parameters can be accessed by their fully qualified names.

Example: Out parameter

```
CREATE VIRTUAL PROCEDURE proc (OUT STRING x, INOUT STRING y) AS
BEGIN
  proc.x = 'some value ' || proc.y;
  y = 'some new value';
END
```

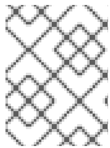
3.8.1.5. Special variables

VARIABLES.ROWCOUNT integer variable will contain the numbers of rows affected by the last INSERT, UPDATE, or DELETE command statement executed. Inserts that are processed by dynamic SQL with an **into** clause will also update the **ROWCOUNT**.

Sample usage

```
...
UPDATE FOO SET X = 1 WHERE Y = 2;
DECLARE INTEGER UPDATED = VARIABLES.ROWCOUNT;
...
```

Non-update command statements (**WITH** or **WITHOUT RETURN**) will reset the **ROWCOUNT** to 0.



NOTE

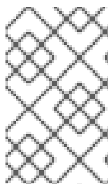
To ensure you are getting the appropriate **ROWCOUNT** value, save the **ROWCOUNT** to a variable immediately after the command statement.

3.8.1.6. Compound statement

A compound statement or block logically groups a series of statements. Temporary tables and variables that are created in a compound statement are local only to that block, and are destroyed when exiting the block.

Usage

```
[label :] BEGIN [[NOT] ATOMIC]
    statement*
[EXCEPTION ex
    statement*
]
END
```



NOTE

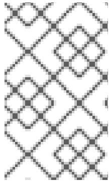
When a block is expected by an **IF**, **LOOP**, **WHILE**, and so forth, a single statement is also accepted by the parser. Even though the block **BEGIN** or **END** are not expected, the statement will execute as if wrapped in a **BEGIN** or **END** pair.

Syntax rules

- If **NOT ATOMIC** or no **ATOMIC** clause is specified, the block will be executed non-atomically.
- If the **ATOMIC** clause is specified, the block must execute atomically. If a transaction is already associated with the thread, no additional action will be taken; savepoints or sub-transactions are not currently used. If the higher level transaction is used, and the block does not complete – regardless of the presence of exception handling – the transaction will be marked as rollback only. Otherwise, a transaction will be associated with the execution of the block. Upon successful completion of the block the transaction will be committed.
- The label must not be the same as any label that is used in statements that contain this one.
- Variable assignments and the implicit result cursor are unaffected by rollbacks. If a block does not complete successfully, its assignments will still take affect.

Exception handling

If an **EXCEPTION** clause is used within a compound statement, any processing exception emitted from statements will be caught with the flow of execution transferring to **EXCEPTION** statements. Any block-level transaction started by this block will commit if the exception handler successfully completes. If another exception, or the original exception, is emitted from the exception handler, the transaction will rollback. Any temporary tables or variables specific to the BLOCK will not be available to the exception handler statements.

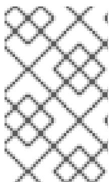


NOTE

Only processing exceptions, which are typically caused by errors originating at the sources or with function execution, are caught. A low-level internal Data Virtualization error or Java **RuntimeException** will not be caught.

To aid in the processing of a caught exception, the **EXCEPTION** clause specifies a group name that exposes the significant fields of the exception. The following table shows the variables that an exception group contains:

Variable	Type	Description
STATE	string	The SQL State
ERRORCODE	integer	The error or vendor code. In the case of Data Virtualization internal exceptions this will be the integer suffix of the TEIIDxxxx code.
TEIIDCODE	string	The full Data Virtualization event code. Typically TEIIDxxxx.
EXCEPTION	object	The exception being caught, will be an instance of **TeiidSQLException**.
CHAIN	object	The chained exception or cause of the current exception.



NOTE

Data Virtualization does not yet fully comply with the ANSI SQL specification on SQL State usage. For Data Virtualization errors without an underlying SQLException cause, it is best to use the Data Virtualization code.

The exception group name might not be the same as any higher level exception group or loop cursor name.

Example exception group handling

```
BEGIN
   DECLARE EXCEPTION e = SQLEXCEPTION 'this is bad' SQLSTATE 'xxxxx';
   RAISE variables.e;
EXCEPTION e
   IF (e.state = 'xxxxx')
      //in this trivial example, we'll always hit this branch and just log the exception
      RAISE SQLWARNING e.exception;
   ELSE
      RAISE e.exception;
END
```

3.8.1.7. IF statement

An IF statement evaluates a condition and executes either one of two statements depending on the result. You can nest IF statements to create complex branching logic. A dependent ELSE statement will execute its statement only if the IF statement evaluates to **false**.

Usage

```
IF (criteria)
   block
[ELSE
   block]
END
```

Example IF statement

```
IF ( var1 = 'North America')
BEGIN
  ...statement...
END ELSE
BEGIN
  ...statement...
END
```

The criteria can be any valid Boolean expression or an **IS DISTINCT FROM** predicate referencing row values. The **IS DISTINCT FROM** extension uses the following syntax:

```
rowVal IS [NOT] DISTINCT FROM rowValOther
```

Where **rowVal** and **rowValOther** are references to row value group. This would typically be used in instead of update triggers on views to quickly determine if the row values are changing:

Example: IS DISTINCT FROM IF statement

```
IF ( "new" IS DISTINCT FROM "old")
BEGIN
  ...statement...
END
```

IS DISTINCT FROM considers null values equivalent and never produces an UNKNOWN value.

TIP

Null values should be considered in the criteria of an IF statement. **IS NULL** criteria can be used to detect the presence of a null value.

3.8.1.8. Loop Statement

A LOOP statement is an iterative control construct that is used to cursor through a result set.

Usage

```
[label :] LOOP ON <select statement> AS <cursorname>
    statement
```

Syntax rules

- The label must not be the same as any label that is used in statements that contain this one.

3.8.1.9. While statement

A **WHILE** statement is an iterative control construct that is used to execute a statement repeatedly whenever a specified condition is met.

Usage

```
[label :] WHILE <criteria>
    statement
```

Syntax rules

- The label must not be the same as any label that is used in statements that contain this one.

3.8.1.10. Continue statement

A **CONTINUE** statement is used inside a **LOOP** or **WHILE** construct to continue with the next loop by skipping over the rest of the statements in the loop. It must be used inside a **LOOP** or **WHILE** statement.

Usage

```
CONTINUE [label];
```

Syntax rules

- If the label is specified, it must exist on a containing **LOOP** or **WHILE** statement.
- If no label is specified, the statement will affect the closest containing **LOOP** or **WHILE** statement.

3.8.1.11. Break statement

A **BREAK** statement is used inside a **LOOP** or **WHILE** construct to break from the loop. It must be used inside a **LOOP** or **WHILE** statement.

Usage

```
BREAK [label];
```

Syntax rules

- If the label is specified, it must exist on a containing **LOOP** or **WHILE** statement.
- If no label is specified, the statement will affect the closest containing **LOOP** or **WHILE** statement.

3.8.1.12. Leave statement

A **LEAVE** statement is used inside a compound, **LOOP**, or **WHILE** construct to leave to the specified level.

Usage

```
LEAVE label;
```

Syntax rules

- The label must exist on a containing compound statement, **LOOP**, or **WHILE** statement.

3.8.1.13. Return statement

A **RETURN** statement gracefully exits the procedure and optionally returns a value.

Usage

```
RETURN [expression];
```

Syntax rules

- If an expression is specified, the procedure must have a return parameter and the value must be implicitly convertible to the expected type.
- Even if the procedure has a return parameter, it is not required to specify a return value in a **RETURN** statement. A return parameter can be set through an assignment or it can be left as null.

Sample usage

```
CREATE VIRTUAL FUNCTION times_two(val integer)
  RETURNS integer AS
  BEGIN
    RETURN val*2;
  END
```

3.8.1.14. Error statement

An **ERROR** statement declares that the procedure has entered an error state and should abort. This statement will also roll back the current transaction, if one exists. Any valid expression can be specified after the **ERROR** keyword.

Usage

```
ERROR message;
```

Example: Error statement

```
ERROR 'Invalid input value: ' || nvl(Acct.GetBalance.AcctID, 'null');
```

An **ERROR** statement is equivalent to:

```
RAISE SQLEXCEPTION message;
```

3.8.1.15. Raise statement

A **RAISE** statement is used to raise an exception or warning. When raising an exception, this statement will also roll back the current transaction, if one exists.

Usage

```
RAISE [SQLWARNING] exception;
```

Where exception may be a variable reference to an exception or an exception expression.

Syntax rules

- If **SQLWARNING** is specified, the exception will be sent to the client as a warning and the procedure will continue to execute.
- A null warning will be ignored. A null non-warning exception will still cause an exception to be raised.

Example raise statement

```
RAISE SQLWARNING SQLEXCEPTION 'invalid' SQLSTATE '05000';
```

3.8.1.16. Exception expression

An exception expression creates an exception that can be raised or used as a warning.

Usage

```
SQLEXCEPTION message [SQLSTATE state [, code]] CHAIN exception
```

Syntax rules

- Any of the values may be null.
- **message** and **state** are string expressions that specify the exception message and SQL state. Data Virtualization does not fully comply with the ANSI SQL specification on SQL state usage, but you are allowed to set any SQL state you choose.
- **code** is an integer expression that specifies the vendor code.
- **exception** must be a variable reference to an exception or an exception expression, and will be chained to the resulting exception as its parent.

3.8.2. Virtual procedures

Virtual procedures are defined using the Data Virtualization procedural language. For more information, see Procedure language.

A virtual procedure has zero or more INPUT, INOUT, or OUT parameters, an optional RETURN parameter, and an optional result set. Virtual procedures can execute queries and other SQL commands, define temporary tables, add data to temporary tables, walk through result sets, use loops, and use conditional logic.

Virtual procedure definition

For more information, see *Create procedure/function* in DDL metadata for schema objects.

Note that the optional result parameter is always considered the first parameter.

Within the body of the procedure, you can use any valid statement. For more information avbout procedure language statements, see Procedure language.

There is no explicit cursoring or value returning statement. Instead, the last unnamed command statement executed in the procedure that returns a result set will be returned as the result. The output of that statement must match the expected result set and parameters of the procedure.

Virtual procedure parameters

Virtual procedures can take zero or more **IN** or **INOUT** parameters, and can have any number of **OUT** parameters and an optional **RETURN** parameter. Each input has the following information that is used during runtime processing:

Name

The name of the input parameter.

Datatype

The design-time type of the input parameter.

Default value

The default value if the input parameter is not specified.

Nullable

NO_NULLS, **NULLABLE**, **NULLABLE_UNKNOWN**; parameter is optional if nullable, and is not required to be listed when using named parameter syntax.

You reference a parameter in a virtual procedure by using its fully-qualified name (or less if unambiguous). For example, **MySchema.MyProc.Param1**.

Example: Referencing an input parameter and assigning an Out parameter for GetBalance procedure

```
BEGIN
  MySchema.GetBalance.RetVal = UPPER(MySchema.GetBalance.AcctID);
  SELECT Balance FROM MySchema.Accts WHERE MySchema.Accts.AccountID =
MySchema.GetBalance.AcctID;
END
```

If an **INOUT** parameter is not assigned any value in a procedure, it will retain the value it was assigned for input. Any **OUT**/**RETURN** parameter that is not assigned a value will retain the default NULL value. The **INOUT**/**OUT**/**RETURN** output values are validated against the **NOT NULL** metadata of the parameter.

Example virtual procedures

The following example represents a loop that walks through a cursored table and uses **CONTINUE** and **BREAK**.

Virtual procedure using LOOP, CONTINUE, BREAK

```
BEGIN
 DECLARE double total;
 DECLARE integer transactions;
 LOOP ON (SELECT amt, type FROM CashTxnTable) AS txncursor
 BEGIN
  IF(txncursor.type <> 'Sale')
  BEGIN
   CONTINUE;
  END ELSE
  BEGIN
   total = (total + txncursor.amt);
   transactions = (transactions + 1);
   IF(transactions = 100)
   BEGIN
    BREAK;
   END
  END
 END
 SELECT total, (total / transactions) AS avg_transaction;
END
```

The following example uses conditional logic to determine which of two SELECT statements to execute.

Virtual procedure with conditional SELECT

```
BEGIN
 DECLARE string VARIABLES.SORTDIRECTION;
 VARIABLES.SORTDIRECTION = PartsVirtual.OrderedQtyProc.SORTMODE;
 IF ( ucase(VARIABLES.SORTDIRECTION) = 'ASC' )
 BEGIN
  SELECT * FROM PartsVirtual.SupplierInfo WHERE QUANTITY >
PartsVirtual.OrderedQtyProc.QTYIN ORDER BY PartsVirtual.SupplierInfo.PART_ID;
 END ELSE
 BEGIN
  SELECT * FROM PartsVirtual.SupplierInfo WHERE QUANTITY >
PartsVirtual.OrderedQtyProc.QTYIN ORDER BY PartsVirtual.SupplierInfo.PART_ID DESC;
 END
END
```

Executing virtual procedures

You execute procedures using the SQL **EXECUTE** command. For more information, see *Execute command* in DML commands.

If the procedure has defined inputs, you specify those in a sequential list, or using *name=value* syntax. You must use the name of the input parameter, scoped by the full procedure name if the parameter name is ambiguous in the context of other columns or variables in the procedure.

A virtual procedure call returns a result set like any **SELECT**, so you can use this in many places you can use a **SELECT**. Typically you'll use the following syntax:

```
SELECT * FROM (EXEC ...) AS x
```

Virtual procedure limitations

A virtual procedure can return only one result set. If you need to pass in a result set, or pass out multiple result sets, then consider using global temporary tables instead.

3.8.3. Triggers

View triggers

Views are abstractions above physical sources. They typically union or join information from multiple tables, often from multiple data sources or other views. Data Virtualization can perform update operations against views. Update commands that you run against a view (**INSERT**, **UPDATE**, or **DELETE**) require logic to define how the tables and views integrated by the view are affected by each type of command. This transformation logic, also referred to as a **trigger**, is invoked when an update command is issued against a view. Update procedures define the logic for how the update command that you run against a view is decomposed into the individual commands to be executed against the underlying physical sources. Similar to virtual procedures, update procedures have the ability to execute queries or other commands, define temporary tables, add data to temporary tables, walk through result sets, use loops, and use conditional logic. For more inmformation about virtual procedures, see Virtual procedures.

You can use **INSTEAD OF** triggers on views in a way that is similar to the way that you might use them with traditional databases. You can have only one **FOR EACH ROW** procedure for each **INSERT**, **UPDATE**, or **DELETE** operation against a view.

Usage

```
CREATE TRIGGER ON view_name INSTEAD OF INSERT|UPDATE|DELETE AS
FOR EACH ROW
...
```

Update procedure processing

1. The user application submits the SQL command.
2. The command detects the view that it is executed against.
3. The correct procedure is chosen depending upon the command type (**INSERT**, **UPDATE**, or **DELETE**).
4. The procedure is executed. The procedure might contain SQL commands of its own. Commands in the procedure can be different in type from the command that is received from the calling application.
5. Commands, as described in the procedure, are issued to the individual physical data sources or other views.
6. A value representing the number of rows changed is returned to the calling application.

Source triggers

Data Virtualization can use **AFTER** triggers on source tables. **AFTER** triggers are called by events from a change data capture (CDC) system.

Usage:

```
CREATE TRIGGER ON source_table AFTER INSERT|UPDATE|DELETE AS
FOR EACH ROW
...
```

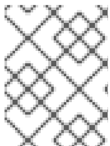
FOR EACH ROW triggers

Only the **FOR EACH ROW** construct serves as a trigger handler. A **FOR EACH ROW** trigger procedure will evaluate its block for each row of the view/source affected by the **UPDATE** statement. For **UPDATE** and **DELETE** statements, this will be every row that passes the **WHERE** condition. For **INSERT** statements there will be one new row for each set of values from the **VALUES** or query expression. For a view, the rows updated is reported as this number, regardless of the affect of the underlying procedure logic.

Usage

```
FOR EACH ROW
  BEGIN ATOMIC
    ...
  END
```

The **BEGIN** and **END** keywords are used to denote block boundaries. Within the body of the procedure, any valid statement may be used.



NOTE

The use of the **ATOMIC** keyword is currently optional for backward compatibility, but unlike a normal block, the default for **INSTEAD OF** triggers is atomic.

Special variables for update procedures

You can use a number of special variables when defining your update procedure.

NEW variables

Every attribute in the view/table whose **UPDATE** and **INSERT** transformations you are defining has an equivalent variable named **NEW.<column_name>**.
When an INSERT or an UPDATE command is executed against the view, or the event is received, these variables are initialized to the values in the **INSERT VALUES** clause or the **UPDATE SET** clause respectively.

In an **UPDATE** procedure, the default value of these variables, if they are not set by the command, is the old value. In an INSERT procedure, the default value of these variables is the default value of the virtual table attributes. See **CHANGING** variables, later in this list for distinguishing defaults from passed values.

OLD variables

Every attribute on the view/table whose **UPDATE** and **DELETE** transformations you are defining has an equivalent variable named **OLD.<column_name>**.
When a **DELETE** or **UPDATE** command is executed against the view, or the event is received, these variables are initialized to the current values of the row being deleted or updated respectively.

CHANGING variables

Every attribute on the view/table whose **UPDATE** and **INSERT** transformations you are defining has an equivalent variable named **CHANGING.<column_name>**.

When an **INSERT** or an **UPDATE** command is executed against the view, or an the event is received, these variables are initialized to **true** or **false** depending on whether the **INPUT** variable was set by the command. A **CHANGING** variable is commonly used to differentiate between a default insert value and one that is specified in the user query.

For example, for a view with columns A, B, C:

If User Executes...	Then...
INSERT INTO VT (A, B) VALUES (0, 1)	CHANGING.A = true, CHANGING.B = true, CHANGING.C = false
UPDATE VT SET C = 2	CHANGING.A = false, CHANGING.B = false, CHANGING.C = true

Key variables

To return generated keys from an **INSERT** trigger, a KEY group is made available that can be assigned the value to be returned. Typically this requires using the **generated_key** system function. However, not all inserts provide generated keys, because not all sources return generated keys.

```
create view v1 (i integer, k integer not null auto_increment primary key)
OPTIONS (UPDATABLE true) as
  select x, y from tbl;
create trigger on v1 instead of insert as
  for each row begin atomic
    -- ... some logic
    insert into tbl (x) values (new.i);
    key.k = cast(generated_key('y') as integer);
  end;
```

Example update procedures

For example, for a view with columns A, B, C:

Sample DELETE procedure

```
FOR EACH ROW
BEGIN
   DELETE FROM X WHERE Y = OLD.A;
   DELETE FROM Z WHERE Y = OLD.A; // cascade the delete
END
```

Sample UPDATE procedure

```
FOR EACH ROW
BEGIN
   IF (CHANGING.B)
   BEGIN
      UPDATE Z SET Y = NEW.B WHERE Y = OLD.B;
   END
END
```

Other usages

FOR EACH ROW update procedures in a view can also be used to emulate **BEFORE**/**AFTER** each row triggers while still retaining the ability to perform an inherent update. This **BEFORE**/**AFTER** trigger behavior with an inherent update can be achieved by creating an additional updatable view over the target view with update procedures of the form:

```
CREATE TRIGGER ON outerVW INSTEAD OF INSERT AS
FOR EACH ROW
  BEGIN ATOMIC
  --before row logic
  ...

  --default insert/update/delete against the target view
  INSERT INTO VW (c1, c2, c3) VALUES (NEW.c1, NEW.c2, NEW.c3);

  --after row logic
  ...
  END
```

3.9. COMMENTS

You can add multi-line SQL comments in Data Virtualization by enclosing text with /* */.

```
/* comment
comment
comment... */
```

You can also add single line comments:

```
SELECT ... -- comment
```

You can also nest comments.

3.10. EXPLAIN STATEMENTS

You can use an EXPLAIN statement to obtain a query plan. Using EXPLAIN statements to obtain a query execution plan is a native function of the SQL language, and it is the preferred mechanism to use over pg/ODBC transport. If you are using a Teiid JDBC client, you can also use SET/SHOW statements. For more information about SET and SHOW statements, see the Client Developer's Guide.

Usage

```
EXPLAIN [(explainOption [, ...])] statement

explainOption :=
    ANALYZE [TRUE|FALSE]
  | FORMAT {TEXT|YAML|XML}
```

If no options are specified, by default the plan is provided in text format without executing the query.

If you specify **ANALYZE** or **ANALYZE TRUE**, then the statement is executed, unless the client has set the **NOEXEC** option. The resulting plan will include runtime node statistics from the fully executed

statement. All side effects, including updates, will still occur. You might need to use a transaction to rollback any unwanted side effects.

While this is superficially the same syntax as PostgreSQL, the plan provided in the various formats is the same that has been provided by Teiid in prior versions.

For more information about how to interpret results, see Query plans.

Example

```
EXPLAIN (analyze) select * from really_complicated_view
```

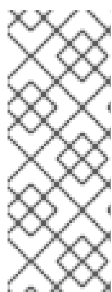
Returns a text-formatted plan from an actual run of the given statement.

CHAPTER 4. DATA TYPES

The Data Virtualization type system is based on Java/JDBC types. The runtime object is represented by the corresponding Java class, such as Long, Integer, Boolean, String, and so forth. For more information, see Runtime types. You can use domain types to extend the type system. For more information, see DDL metadata for domains.

4.1. RUNTIME TYPES

Data Virtualization works with a core set of runtime types. Runtime types can be different from semantic types that are defined in type fields at design time. The runtime type can also be specified at design time or it will be automatically chosen as the closest base type to the semantic type.



NOTE

Even if a type is declared with a length, precision, or scale argument, those restrictions are effectively ignored by the runtime system, but may be enforced/reported at the edge by OData, ODBC, JDBC. Geospatial types act in a similar manner. Extension metadata might be needed for SRID, type, and number of dimensions for consumption by tools/OData, but it is not yet enforced. In some instances you might need to use the ST_SETSRID function to ensure the SRID is associated.

Table 4.1. Data Virtualization runtime types

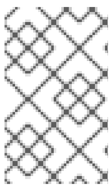
Type	Description	Java runtime class	JDBC type	ODBC type
string or varchar	Variable length character string with a maximum length of 4000.	java.lang.String	VARCHAR	VARCHAR
varbinary	Variable length binary string with a nominal maximum length of 8192.	byte[] [1]	VARBINARY	VARBINARY
char	A single 16 bit character - which cannot represent a value beyond the Basic Multilingual Plane. This limitation also applies to functions/expressions that expect a single character such as trim, textagg, texttable, and like escape.	java.lang.Character	CHAR	CHAR

Type	Description	Java runtime class	JDBC type	ODBC type
boolean	A single bit, or Boolean, that can be true, false, or null (unknown)	java.lang.Boolean	BIT	SMALLINT
byte or tinyint	Numeric, integral type, signed 8-bit	java.lang.Byte	TINYINT	SMALLINT
short or smallint	Numeric, integral type, signed 16-bit	java.lang.Short	SMALLINT	SMALLINT
integer or serial	Numeric, integral type, signed 32-bit. The serial type also implies not null and has an auto-incrementing value that starts at 1. serial types are not automatically UNIQUE.	java.lang.Integer	INTEGER	INTEGER
long or bigint	Numeric, integral type, signed 64-bit	java.lang.Long	BIGINT	NUMERIC
biginteger	Numeric, integral type, arbitrary precision of up to 1000 digits	java.math.BigInteger	NUMERIC	NUMERIC
float or real	Numeric, floating point type, 32-bit IEEE 754 floating-point numbers	java.lang.Float	REAL	FLOAT
double	Numeric, floating point type, 64-bit IEEE 754 floating-point numbers	java.lang.Double	DOUBLE	DOUBLE
bigdecimal or decimal	Numeric, floating point type, arbitrary precision of up to 1000 digits.	java.math.BigDecimal	NUMERIC	NUMERIC

Type	Description	Java runtime class	JDBC type	ODBC type
date	Datetime, representing a single day (year, month, day)	java.sql.Date	DATE	DATE
time	Datetime, representing a single time (hours, minutes, seconds)	java.sql.Time	TIME	TIME
timestamp	Datetime, representing a single date and time (year, month, day, hours, minutes, seconds, fractional seconds).	java.sql.Timestamp	TIMESTAMP	TIMESTAMP
object	Any arbitrary Java object, must implement java.lang.Serializable.	Any	JAVA_OBJECT	VARCHAR
blob	Binary large object, representing a stream of bytes.	java.sql.Blob [2]	BLOB	VARCHAR
clob	Character large object, representing a stream of characters.	java.sql.Clob [3]	CLOB	VARCHAR
xml	XML document	java.sql.SQLXML[4]	JAVA_OBJECT	VARCHAR
geometry	Geospatial Object	java.sql.Blob [5]	BLOB	BLOB
geography (11.2+)	Geospatial Object	java.sql.Blob [6]	BLOB	BLOB

Type	Description	Java runtime class	JDBC type	ODBC type
json (11.2+)	Character large object, representing a stream of JSON characters.	java.sql.Clob [7]	CLOB	VARCHAR

1. The runtime type is org.teiid.core.types.BinaryType. Translators will need to explicitly handle BinaryType values. UDFs will instead have a byte[] value passed.
2. The concrete type is expected to be org.teiid.core.types.BlobType
3. The concrete type is expected to be org.teiid.core.types.ClobType
4. The concrete type is expected to be org.teiid.core.types.XMLType
5. The concrete type is expected to be org.teiid.core.types.GeometryType
6. The concrete type is expected to be org.teiid.core.types.GeographyType
7. The concrete type is expected to be org.teiid.core.types.JsonType



NOTE

Character, String, and character large objects (CLOB) types are not limited to ASCII/extended ASCII values. Character can hold codes up to 2^16-1 and String/CLOB can hold any value.

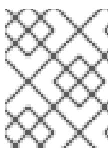
Arrays

An array of any type is designated by adding [] for each array dimension to the type declaration.

Example: Array types

```
string[]
```

```
integer[][]
```



NOTE

Array handling is typically in memory. It is not advisable to rely on the usage of large array values. Arrays of large objects (LOBs) are typically not handled correctly when serialized.

4.2. TYPE CONVERSIONS

Data types may be converted from one form to another either explicitly or implicitly. Implicit conversions automatically occur in criteria and expressions to ease development. Explicit datatype conversions require the use of the **CONVERT** function or **CAST** keyword.

Type conversion considerations

- Any type may be implicitly converted to the **OBJECT** type.
- The **OBJECT** type can be explicitly converted to any other type.
- The NULL value can be converted to any type.
- Any valid implicit conversion is also a valid explicit conversion.
- In scenarios where literal values would normally require explicit conversions, you can apply implicit conversions if no loss of information occurs.
- If **widenComparisonToString** is false (the default), Data Virtualization raises an exception if it detects that an explicit conversion cannot be applied implicitly in criteria.
- If **widenComparisonToString** is true, then depending upon the comparison, a widening conversion is applied or the criteria are treated as false. For more information about **widenComparisonToString**, see *System properties* in the Administrator's Guide.

  Example

  ```
  SELECT * FROM my.table WHERE created_by = 'not a date'
  ```

  If **widenComparisonToString** is false, and **created_by** is a date, **not a date** cannot be converted to a date value, and an exception results.
- Explicit conversions that are not allowed between two types will result in an exception before execution. Allowed explicit conversions can still fail during processing if the runtime values are not actually convertible.

WARNING

The Data Virtualization conversions of float/double/bigdecimal/timestamp to string rely on the JDBC/Java defined output formats. Pushdown behavior attempts to mimic these results, but can vary depending upon the actual source type and conversion logic. It is best not to assume use of the string form in criteria or other places where variations might lead to different results.

Table 4.2. Type conversions

Source type	Valid implicit target types	Valid explicit target types
string	clob	char, boolean, byte, short, integer, long, biginteger, float, double, bigdecimal, xml [a]
char	string	

Source type	Valid implicit target types	Valid explicit target types
boolean	string, byte, short, integer, long, biginteger, float, double, bigdecimal	
byte	string, short, integer, long, biginteger, float, double, bigdecimal	boolean
short	string, integer, long, biginteger, float, double, bigdecimal	boolean, byte
integer	string, long, biginteger, double, bigdecimal	boolean, byte, short, float
long	string, biginteger, bigdecimal, float [b], double [b]	boolean, byte, short, integer, float, double
biginteger	string, bigdecimal float [b], double [b]	boolean, byte, short, integer, long, float, double
bigdecimal	string, float [b], double [b]	boolean, byte, short, integer, long, biginteger, float, double
float	string, bigdecimal, double	boolean, byte, short, integer, long, biginteger
double	string, bigdecimal, float [b]	boolean, byte, short, integer, long, biginteger, float
date	string, timestamp	
time	string, timestamp	
timestamp	string	date, time
clob		string
json	clob	string
xml		string [c]
geography		geometry

Source type	Valid implicit target types	Valid explicit target types

[a] string to xml is equivalent to XMLPARSE(DOCUMENT exp). For more information, see *XMLPARSE* in XML functions.

[b] Implicit conversion to float/double only occurs for literal values.

[c] xml to string is equivalent to XMLSERIALIZE(exp AS STRING). For more information, see *XMLSERIALIZE* in XML functions.

4.3. SPECIAL CONVERSION CASES

Conversion of string literals

Data Virtualization automatically converts string literals within a SQL statement to their implied types. This typically occurs in a criteria comparison where an expression with a different datatype is compared to a literal string. For example:

```
SELECT * FROM my.table WHERE created_by = '2016-01-02'
```

In the preceding example, if the **created_by** column has the data type of date, Data Virtualization automatically converts the data type of the string literal to a date.

Converting to Boolean

Data Virtualization can automatically convert literal strings and numeric type values to Boolean values as shwon in the following table:

Table 4.3. Boolean conversions

Type	Literal value	Boolean value
String	'false'	false
	'unknown'	null
	other	true
Numeric	0	false
	other	true

Date and time conversions

Data Virtualization can implicitly convert properly formatted literal strings to their associated date-related data types as shown in the following table:

Table 4.4. Date and time conversions

String literal format	Possible implicit conversion type
yyyy-mm-dd	DATE

String literal format	Possible implicit conversion type
hh:mm:ss	TIME
yyyy-mm-dd[hh:mm:ss.[fff...]]	TIMESTAMP

The preceding formats are those expected by the JDBC date types. For information about using other formats, see the functions **PARSEDATE**, **PARSETIME**, and **PARSETIMESTAMP** in Date and time functions.

4.4. ESCAPED LITERAL SYNTAX

Rather than relying on implicit conversion, you can define data type values directly in SQL by using escape syntax. The string values that you supply must match the expected format exactly, or an exception will occur.

Datatype	Escaped syntax	Standard literal
BOOLEAN	{b 'true'}	TRUE
DATE	{d 'yyyy-mm-dd'}	DATE 'yyyy-mm-dd'
TIME	{t 'hh-mm-ss'}	TIME 'hh-mm-ss'
TIMESTAMP	{ts 'yyyy-mm-dd[hh:mm:ss.[fff...]]'}	TIMESTAMP 'yyyy-mm-dd[hh:mm:ss.[fff...]]'

CHAPTER 5. UPDATABLE VIEWS

Any view can be marked as *updatable*. In many circumstances the view definition allows the view to be inherently updatable without the need to manually define a trigger to handle **INSERT**/**UPDATE**/**DELETE** operations.

An inherently updatable view cannot be defined with a query that has:

- A set operation (**INTERSECT**, **EXCEPT**, **UNION**).
- **SELECT DISTINCT**.
- Aggregation (aggregate functions, **GROUP BY**, **HAVING**).
- A **LIMIT** clause.

A **UNION ALL** can define an inherently updatable view only if each of the UNION branches are themselves inherently updatable. A view defined by a **UNION ALL** can accommodate inherent **INSERT** statements if it is a partitioned union, and the **INSERT** specifies values that belong to a single partition. For more information, see *partitioned union* in Federated optimizations.

Any view column that is not mapped directly to a column is not updatable and cannot be targeted by an UPDATE set clause or be an INSERT column.

If a view is defined by a join query or has a **WITH** clause it might still be inherently updatable. However, in these situations there are further restrictions, and the resulting query plan may execute multiple statements. For a non-simple query to be updatable, the following criteria apply:

- An **INSERT/UPDATE** can only modify a single key-preserved table.
- To allow **DELETE** operations, there must be only a single key-preserved table.
  For information about key-preserved tables, see Key-preserved tables.

If the default handling is not available or if you want to have an alternative implementation of an **INSERT/UPDATE/DELETE**, you can use update procedures, or triggers, to define procedures to handle the respective operations. For more information see Update procedures (Triggers) .

Consider the following example of an inherently updatable denormalized view:

```
create foreign table parent_table (pk_col integer primary key, name string) options (updatable true);

create foreign table child_table (pk_col integer primary key, name string, fk_col integer, foreign key
(fk_col) references parent_table (pk_col)) options (updatable true);

create view denormalized options (updatable true) as select c.fk_col, c.name as child_name, p.name
from parent_table as p, child_table as c where p.pk_col = c.fk_col;
```

A query such as **insert into denormalized (fk_col, child_name) values (1, 'a')** would succeed against this view, because it targets a single key-preserved table, **child_table**. However, **insert into denormalized (name) values ('a')** would fail, because it maps to a **parent_table** that can have multiple rows for each **parent_table** key. In other words, it is not *key-preserved*.

Also, an **INSERT** against **parent_table** alone might not be visible to the view, because there might be no child entities associated either.

Not all scenarios will work. Referencing the preceding example, an **insert into denormalized (pk_col, child_name) values (1, 'a')** with a view that is defined using the **p.pk_col** will fail, because the logic doesn't yet consider the equivalency of the key values.

5.1. KEY-PRESERVED TABLES

A key-preserved table has a primary or unique key that remains unique when it is projected into the result of the query. Note that it is not actually required for a view to reference the key columns in the SELECT clause. The query engine can detect a key-preserved table by analyzing the join structure. The engine will ensure that a join of a key-preserved table must be against one of its foreign keys.

CHAPTER 6. TRANSACTIONS

Data Virtualization utilizes XA transactions for participating in global transactions and for demarcating its local and command scoped transactions.

For information about advanced transaction technologies that are provided for Data Virtualization through the Narayana community project, see the Narayana documentation.

Table 6.1. Data Virtualization transaction scopes

Scope	Description
Command	Treats the user command as if all source commands are executed within the scope of the same transaction. The AutoCommitTxn execution property controls the behavior of command level transactions.
Local	The transaction boundary is local defined by a single client session.
Global	Data Virtualization participates in a global transaction as an XA resource.

The default transaction isolation level for Data Virtualization is READ_COMMITTED.

6.1. AUTOCOMMITTXN EXECUTION PROPERTY

User level commands can execute multiple source commands. To control the transactional behavior of a user command when not in a local or global transaction, you can specify the AutoCommitTxn execution property.

Table 6.2. AutoCommitTxn Settings

Setting	Description
OFF	Do not wrap each command in a transaction. Individual source commands may commit or rollback regardless of the success or failure of the overall command.
ON	Wrap each command in a transaction. This mode is the safest, but may introduce performance overhead.
DETECT	This is the default setting. Will automatically wrap commands in a transaction, but only if the command seems to be transactionally unsafe.

The concept of command safety with respect to a transaction is determined by Data Virtualization based upon command type, the transaction isolation level, and available metadata. A wrapping transaction is not needed if the following criteria are true:

- The user command is fully pushed to the source.
- The user command is a SELECT (including XML) and the transaction isolation is not REPEATABLE_READ nor SERIALIABLE.
- The user command is a stored procedure, the transaction isolation is not REPEATABLE_READ nor SERIALIABLE, and the updating model count is zero. For more information, see Updating model count.

The update count may be set on all procedures as part of the procedure metadata in the model.

6.2. UPDATING MODEL COUNT

The term "updating model count" refers to the number of times any model is updated during the execution of a command. It is used to determine whether a transaction, of any scope, is required to safely execute the command.

Table 6.3. Updating model count settings

Count	Description
0	No updates are performed by this command.
1	Indicates that only one model is updated by this command (and its subcommands). The success or failure of that update corresponds to the success or failure of the command. It should not be possible for the update to succeed while the command fails. Execution is not considered transactionally unsafe.
*	Any number greater than 1 indicates that execution is transactionally unsafe and an XA transaction will be required.

6.3. JDBC AND TRANSACTIONS

JDBC API functionality

The transaction scopes in Transactions map to the following JDBC modes:

Command

Connection autoCommit property set to true.

Local

Connection autoCommit property set to false. The transaction is committed by setting autoCommit to true or calling **java.sql.Connection.commit**. The transaction can be rolled back by a call to **java.sql.Connection.rollback**

Global

The XAResource interface provided by an XAConnection is used to control the transaction. Note that XAConnections are available only if Data Virtualization is consumed through its XADataSource, **org.teiid.jdbc.TeiidDataSource**. JEE containers or data access APIs typically control XA transactions on behalf of application code.

J2EE usage models

J2EE provides the following ways to manage transactions for beans:

Client-controlled

The client of a bean begins and ends a transaction explicitly.

Bean-managed

The bean itself begins and ends a transaction explicitly.

Container-managed

The application server container begins and ends a transaction automatically.

In any of the preceding cases, transactions can be either local or XA transactions, depending on how the code and descriptors are written. The XA specification does not require some types of beans (for example, stateful session beans and entity beans) to work with non-transactional sources. However, according to the specification, optionally, application servers can allow the use of these beans with non-transactional sources, with the caution that such usage is not portable or predictable. Generally speaking, to provide for most types of EJB activities in a portable fashion, applications require a mechanism for managing transactions.

6.4. LIMITATIONS

- The client setting of transaction isolation level is propagated only to JDBC connectors; the setting is not propagated to other connector types. The default transaction isolation level can be set on each XA connector. However, the isolation level is fixed, and cannot be changed at runtime for specific connections or commands.

CHAPTER 7. DATA ROLES

Data roles, also called entitlements, are sets of permissions defined per virtual database that specify data access permissions (create, read, update, delete). Data roles use a fine-grained permission system that Data Virtualization will enforce at runtime and provide audit log entries for access violations.

Before you apply data roles, you might want to restrict source system access through the fundamental design of your virtual database. Foremost, Data Virtualization can only access source entries that are represented in imported metadata. You should narrow imported metadata to only what is necessary for use by your virtual database.

If data role validation is enabled and data roles are defined in a virtual database, then access permissions will be enforced by the Data Virtualization server. The use of data roles may be disabled system wide by removing the setting for the **teiid** subsystem policy-decider-module. Data roles also have built-in security functions that can be used for row-based and other authorization checks.

WARNING

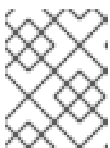
A virtual database that is deployed without data roles can be accessed by any authenticated user.

TIP

By default, non-hidden schema metadata is only visible over JDBC/pg if the user is permissioned in some way for the given object. OData access provides all non-hidden metadata by default. To configure JDBC/pg to also make all non-hidden schema metadata visible to all authenticated users, set the environment/system property **org.teiid.metadataRequiresPermission** to false.

7.1. PERMISSIONS

Permissions, or grants, control access to data in several ways. There are simple access restrictions to SELECT, UPDATE, and so forth, down to a column level.



NOTE

Column or table metadata are not visible to JDBC/ODBC users unless the user has permission to read at least a single column.

You may also use permissions to filter and mask results, and constrain/check update values.

User query permissions

CREATE, READ, UPDATE, DELETE (CRUD) permissions can be set for any resource path in a VDB. A resource path can be as specific as the fully qualified name of a column or as general a top level model (schema) name. Permissions granted to a particular path apply to it and any resource paths that share the same partial name. For example, granting select to "model" will also grant select to "model.table", "model.table.column", and so on. Allowing or denying a particular action is determined by searching for permissions from the most to least specific resource paths. The first permission found with a specific allow or deny will be used. Thus, it is possible to set very general permissions at high-level resource path names and to override only as necessary at more specific resource paths.

Permission grants are only needed for resources that a role needs access to. Permissions are also applied only to the columns/tables/procedures in the user query, not to every resource that is accessed transitively through view and procedure definitions. It is important therefore to ensure that permission grants are applied consistently across models that access the same resources.

WARNING

Non-visible models are accessible by user queries. To restrict user access at a model level, at least one data role should be created to enable data role checking. In turn, that role can be mapped to any authenticated user, and should not grant permissions to models that should be inaccessible.

Permissions are not applicable to the SYS and pg_catalog schemas. These metadata reporting schemas are always accessible regardless of the user. The SYSADMIN schema however may need permissions as applicable.

Permission assignment

To process a *SELECT* statement or a stored procedure execution, the user account requires the following access rights:

- *SELECT*- on the Table(s) being accessed or the procedure being called.
- *SELECT*- on every column referenced.

To process an *INSERT* statement, the user account requires the following access rights:

- *INSERT*- on the Table being inserted into.
- *INSERT*- on every column being inserted on that Table.

To process an *UPDATE* statement, the user account requires the following access rights:

- *UPDATE*- on the Table being updated.
- *UPDATE*- on every column being updated on that Table.
- *SELECT*- on every column referenced in the criteria.

To process a *DELETE* statement, the user account requires the following access rights:

- *DELETE*- on the Table being deleted.
- *SELECT*- on every column referenced in the criteria.

To process a *EXEC/CALL* statement, the user account requires the following access rights:

- *EXECUTE (or SELECT)*- on the Procedure being executed.

To process any function, the user account requires the following access rights:

- *EXECUTE (or SELECT)*- on the Function being called.

To process any ALTER or CREATE TRIGGER statement, the user account requires the following access rights:

- *ALTER*- on the view or procedure that is effected. INSTEAD OF Triggers (update procedures) are not yet treated as full schema objects and are instead treated as attributes of the view.

To process any OBJECTTABLE function, the user account requires the following access rights:

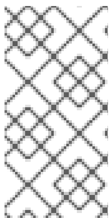
- *LANGUAGE* - specifying the language name that is allowed.

To process any statement against a Data Virtualization temporary table requires the following access rights:

- allow-create-temporary-tables attribute on any applicable role
- *SELECT,INSERT,UPDATE,DELETE* - against the target model/schema as needed for operations against a FOREIGN temporary table.

Row- and column-based security

Although specified in a similar way to user query CRUD permissions, row-based and column-based permissions may be used together or separately to control the data that is returned to users at a more granular and consistent level.



ROW-BASED SECURITY

Specifying a condition on a GRANT for row based security has been deprecated. Specifying a condition on a GRANT is the same as specifying "CREATE POLICY policyName ON schemaName.tblName TO role USING (condition);", such that the condition applies to all operations.

A POLICY against a fully qualified table/view/procedure may specify a condition to be satisfied by the given role. The condition can be any valid boolean expression referencing the columns of the table/view/procedure. Procedure result set columns may be referenced as **proc.col**. The condition will act as a row-based filter and as a checked constraint for insert/update operations.

Application of row-based conditions

A condition is applied conjunctively to update/delete/select WHERE clauses against the affected resource. Those queries will therefore only ever be effective against the subset of rows that pass the condition, such as "SELECT * FROM TBL WHERE something **AND condition**. The condition will be present regardless of how the table/view is used in the query, whether by means of a union, join, or other operation.

Example condition

```
CREATE POLICY policyName ON schemaName.tblName TO superUser USING ('foo=bar');
```

Inserts and updates against physical tables affected by a condition are further validated so that the insert/change values must pass the condition (evaluate to true) for the insert/update to succeed – this is effectively the same a SQL constraint. This will happen for all styles of insert/update – insert with query expression, bulk insert/update, and so on. Inserts/updates against views are not checked with regards to the constraint.

You can disable the insert/update constraint check by restricting the operations that the POLICY applies to.

Example DDL non-constraint condition

```
CREATE POLICY readPolicyName ON schemaName.tblName FOR SELECT,DELETE TO
superUser USING ('col>10');
```

You may of course add another POLICY to cover the INSERT and UPDATE operations should they require a different condition.

If more than one POLICY applies to the same resource, the conditions will be accumulated disjunctively via OR, that is, "(condition1) **OR** (condition2) …". Therefore, creating a POLICY with the condition "true" will allow users in that role to see all rows of the given resource for the given operations.

Considerations when using conditions

Be aware that non-pushdown conditions may adversely impact performance. Avoid using multiple conditions against the same resource as any non-pushdown condition will cause the entire OR statement to not be pushed down. If you need to insert permission conditions, be careful when adding an inline view, because adding them can cause performance problems if they are not compatible with your sources.

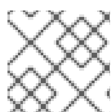
Pushdown of multi-row insert/update operations will be inhibited since the condition must be checked for each row.

You can manage permission conditions on a per-role basis, but another approach is to add condition permissions to any authenticated role. By adding permissions in this way, the conditions are generalized for anyone using the **hasRole**, **user**, and other security functions. The advantage of this latter approach is that it provides you with a static row-based policy. As a result, your entire range of query plans can be shared among your users.

How you handle null values is up to you. You can implement ISNULL checks to ensure that null values are allowed when a column is *nullable*.

Limitations when using conditions

- Conditions on source tables that act as check constraints must currently not contain correlated subqueries.
- Conditions may not contain aggregate or windowed functions.
- Tables and procedures referenced via subqueries will still have row-based filters and column masking applied to them.



NOTE

Row-based filter conditions are enforced even for materialized view loads.

You should ensure that tables consumed to produce materialized views do not have row-based filter conditions on them that could affect the materialized view results.

Column masking

A permission against a fully qualified table/view/procedure column can also specify a mask and optionally a condition. When the query is submitted, the roles are consulted, and the relevant mask/condition information are combined to form a searched case expression to mask the values that would have been returned by the access. Unlike the CRUD allow actions defined above, the resulting

masking effect is always applied – not just at the user query level. The condition and expression can be any valid SQL referencing the columns of the table/view/procedure. Procedure result set columns may be referenced as **proc.col**.

Application of column masks

Column masking is applied only against SELECTs. Column masking is applied logically after the affect of row-based security. However, because both views and source tables canb have row- and column-based security, the actual view-level masking can take place on top of source level masking. If the condition is specified along with the mask, then the effective mask expression affects only a subset of the rows: "CASE WHEN condition THEN mask ELSE column". Otherwise the condition is assumed to be TRUE, meaning that the mask applies to all rows.

If multiple roles specify a mask against a column, the mask order argument will determine their precedence from highest to lowest as part of a larger searched case expression. For example, a mask with the default order of 0 and a mask with an order of 1 would be combined as "CASE WHEN condition1 THEN mask1 WHEN condition0 THEN mask0 ELSE column".

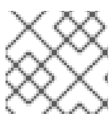
Column masking considerations

Non-pushdown masking conditions/expressions can adversely impact performance, because their evaluation might inhibit pushdown of query constructs on top of the affected resource. In some circumstances the insertion of masking may require that the plan be altered with the addition of an inline view, which can result in poor performance if your sources are not compatible with the use of inline views.

In addition to managing masking on a per-role basis with the use of the order value, another approach is to specify masking in a single any authenticated role such that the conditions/expressions are generalized for all users/roles using the **hasRole**, **user**, and other such security functions. The advantage of the latter approach is that there is effectively a static masking policy in effect, such that all query plans can still be shared between users.

Column masking limitations

- If two masks have the same order value, it is not well defined what order they are applied in.
- Masks or their conditions cannot contain aggregate or windowed functions.
- Tables and procedures referenced via subqueries will still have row-based filters and column masking applied to them.



NOTE

Masking is enforced even for materialized view loads.

You should ensure that tables consumed to produce materialized views do not have masking on them that could affect the materialized view results.

7.2. ROLE MAPPING

Each Data Virtualization data role can be mapped to any number of container roles or to any authenticated user.

It is possible for a user to have any number of container roles, which in turn imply a subset of Data Virtualization data roles. Each applicable Data Virtualization data role contributes cumulatively to the permissions of the user. No one role supersedes or negates the permissions of the other data roles.

CHAPTER 8. SYSTEM SCHEMA

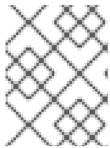
The built-in SYS and SYSADMIN schemas provide metadata tables and procedures against the current virtual database.

By default, a system schema for ODBC metadata pg_catalog is also exposed. – however, that should be considered for general use.

Metadata visibility

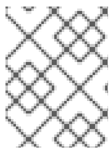
The SYS system schema tables and procedures are always visible and accessible.

When data roles are in use, users can view only the tables, views, and procedure metadata entries that they have permissions to access. All columns of a key must be accessible for an entry to be visible.



NOTE

To make all metadata visible to any authenticated user, set the environment/system property **org.teiid.metadataRequiresPermission** to false.



NOTE

If you use data roles, visibility of entries can be affected by the caching of system metadata.

8.1. SYS SCHEMA

System schema for public information and actions.

SYS.Columns

This table supplies information about all the elements (columns, tags, attributes, etc) in the virtual database.

Column name	Type	Description
VDBName	string	VDB name
SchemaName	string	Schema name
TableName	string	Table name
Name	string	Element name (not qualified)
Position	integer	Position in group (1-based)
NameInSource	string	Name of element in source
DataType	string	Data Virtualization runtime data type name

Column name	Type	Description
Scale	integer	Number of digits after the decimal point
ElementLength	integer	Element length (mostly used for strings)
sLengthFixed	boolean	Whether the length is fixed or variable
SupportsSelect	boolean	Element can be used in SELECT
SupportsUpdates	boolean	Values can be inserted or updated in the element
IsCaseSensitive	boolean	Element is case-sensitive
IsSigned	boolean	Element is signed numeric value
IsCurrency	boolean	Element represents monetary value
IsAutoIncremented	boolean	Element is auto-incremented in the source
NullType	string	Nullability: "Nullable", "No Nulls", "Unknown"
MinRange	string	Minimum value
MaxRange	string	Maximum value
DistinctCount	integer	Distinct value count, -1 can indicate unknown
NullCount	integer	Null value count, -1 can indicate unknown
SearchType	string	Searchability: "Searchable", "All Except Like", "Like Only", Unsearchable"
Format	string	Format of string value
DefaultValue	string	Default value

Column name	Type	Description
JavaClass	string	Java class that will be returned
Precision	integer	Number of digits in numeric value
CharOctetLength	integer	Measure of return value size
Radix	integer	Radix for numeric values
GroupUpperName	string	Upper-case full group name
UpperName	string	Upper-case element name
UID	string	Element unique ID
Description	string	Description
TableUID	string	Parent Table unique ID
TypeName	string	The type name, which may be a domain name
TypeCode	integer	JDBC SQL type code
ColumnSize	string	If numeric, the precision, if character, the length, and if date/time, then the string length of a literal value.

SYS.DataTypes

This table supplies information on datatypes.

Column name	Type	Description
Name	string	Data Virtualization type or domain name
IsStandard	boolean	True if the type is basic
Type	String	One of Basic, UserDefined, ResultSet, Domain
TypeName	string	Design-time type name (same as Name)
JavaClass	string	Java class returned for this type

Column name	Type	Description
Scale	integer	Max scale of this type
TypeLength	integer	Max length of this type
NullType	string	Nullability: "Nullable", "No Nulls", "Unknown"
IsSigned	boolean	Is signed numeric?
IsAutoIncremented	boolean	Is auto-incremented?
IsCaseSensitive	boolean	Is case-sensitive?
Precision	integer	Max precision of this type
Radix	integer	Radix of this type
SearchType	string	Searchability: "Searchable", "All Except Like", "Like Only", "Unsearchable"
UID	string	Data type unique ID
RuntimeType	string	Data Virtualization runtime data type name
BaseType	string	Base type
Description	string	Description of type
TypeCode	integer	JDBC SQL type code
Literal_Prefix	string	literal prefix
Literal_Prefix	string	literal suffix

SYS.KeyColumns

This table supplies information about the columns referenced by a key.

Column name	Type	Description
VDBName	string	VDB name

Column name	Type	Description
SchemaName	string	Schema name
TableName	string	Table name
Name	string	Element name
KeyName	string	Key name
KeyType	string	Key type: "Primary", "Foreign", "Unique", etc
RefKeyUID	string	Referenced key UID
UID	string	Key UID
Position	integer	Position in key
TableUID	string	Parent Table unique ID

SYS.Keys

This table supplies information about primary, foreign, and unique keys.

Column name	Type	Description
VDBName	string	VDB name
SchemaName	string	Schema name
Table name	string	Table name
Name	string	Key name
Description	string	Description
NameInSource	string	Name of key in source system
Type	string	Type of key: "Primary", "Foreign", "Unique", etc
IsIndexed	boolean	True if key is indexed
RefKeyUID	string	Referenced key UID (if foreign key)

Column name	Type	Description
RefTableUID	string	Referenced key table UID (if foreign key)
RefSchemaUID	string	Referenced key table schema UID (if foreign key)
UID	string	Key unique ID
TableUID	string	Key Table unique ID
SchemaUID	string	Key Table Schema unique ID
ColPositions	short[]	Array of column positions within the key table

SYS.ProcedureParams

This supplies information on procedure parameters.

Column name	Type	Description
VDBName	string	VDB name
SchemaName	string	Schema name
ProcedureName	string	Procedure name
Name	string	Parameter name
DataType	string	Data Virtualization runtime data type name
Position	integer	Position in procedure args
Type	string	Parameter direction: "In", "Out", "InOut", "ResultSet", "ReturnValue"
Optional	boolean	Parameter is optional
Precision	integer	Precision of parameter
TypeLength	integer	Length of parameter value
Scale	integer	Scale of parameter

Column name	Type	Description
Radix	integer	Radix of parameter
NullType	string	Nullability: "Nullable", "No Nulls", "Unknown"
Description	string	Description of parameter
TypeName	string	The type name, which may be a domain name
TypeCode	integer	JDBC SQL type code
ColumnSize	string	If numeric, the precision, if character, the length, and if date/time, then the string length of a literal value.
DefaultValue	string	Default value

SYS.Procedures

This table supplies information about the procedures in the virtual database.

Column name	Type	Description
VDBName	string	VDB name
SchemaName	string	Schema name
Name	string	Procedure name
NameInSource	string	Procedure name in source system
ReturnsResults	boolean	Returns a result set
UID	string	Procedure UID
Description	string	Description
SchemaUID	string	Parent Schema unique ID

SYS.FunctionParams

This supplies information on function parameters.

Column name	Type	Description
VDBName	string	VDB name
SchemaName	string	Schema name
FunctionName	string	Function name
FunctionUID	string	Function UID
Name	string	Parameter name
DataType	string	Data Virtualization runtime data type name
Position	integer	Position in procedure args
Type	string	Parameter direction: "In", "Out", "InOut", "ResultSet", "ReturnValue"
Precision	integer	Precision of parameter
TypeLength	integer	Length of parameter value
Scale	integer	Scale of parameter
Radix	integer	Radix of parameter
NullType	string	Nullability: "Nullable", "No Nulls", "Unknown"
Description	string	Description of parameter
TypeName	string	The type name, which may be a domain name
TypeCode	integer	JDBC SQL type code
ColumnSize	string	If numeric, the precision, if character, the length, and if date/time, then the string length of a literal value.

SYS.Functions

This table supplies information about the functions in the virtual database.

Column name	Type	Description
VDBName	string	VDB name
SchemaName	string	Schema name
Name	string	Function name
NameInSource	string	Function name in source system
UID	string	Function UID
Description	string	Description
IsVarArgs	boolean	Does the function accept variable arguments

SYS.Properties

This table supplies user-defined properties on all objects based on metamodel extensions. Normally, this table is empty if no metamodel extensions are being used.

Column name	Type	Description
Name	string	Extension property name
Value	string	Extension property value
UID	string	Key unique ID
ClobValue	clob	Clob Value

SYS.ReferenceKeyColumns

This table supplies informaton about column’s key reference.

Column name	Type	Description
PKTABLE_CAT	string	VDB name
PKTABLE_SCHEM	string	Schema name
PKTABLE_NAME	string	Table/View name
PKCOLUMN_NAME	string	Column name
FKTABLE_CAT	string	VDB name

Column name	Type	Description
FKTABLE_SCHEM	string	Schema name
FKTABLE_NAME	string	Table/View name
FKCOLUMN_NAME	string	Column name
KEY_SEQ	short	Key Sequence
UPDATE_RULE	integer	Update Rule
DELETE_RULE	integer	Delete Rule
FK_NAME	string	FK name
PK_NAME	string	PK Nmae
DEFERRABILITY	integer	

SYS.Schemas

This table supplies information about all the schemas in the virtual database, including the system schema itself (System).

Column name	Type	Description
VDBName	string	VDB name
Name	string	Schema name
IsPhysical	boolean	True if this represents a source
UID	string	Unique ID
Description	string	Description
PrimaryMetamodelURI	string	URI for the primary metamodel describing the model used for this schema

SYS.Tables

This table supplies information about all the groups (tables, views, documents, and so forth) in the virtual database.

Column name	Type	Description
VDBName	string	VDB name
SchemaName	string	Schema Name
Name	string	Short group name
Type	string	Table type (Table, View, Document, ...)
NameInSource	string	Name of this group in the source
IsPhysical	boolean	True if this is a source table
SupportsUpdates	boolean	True if group can be updated
UID	string	Group unique ID
Cardinality	integer	Approximate number of rows in the group
Description	string	Description
IsSystem	boolean	True if in system table
SchemaUID	string	Parent Schema unique ID

SYS.VirtualDatabases

This table supplies information about the currently connected virtual database, of which there is always exactly one (in the context of a connection).

Column name	Type	Description
Name	string	The name of the VDB
Version	string	The version of the VDB
Description	string	The description of the VDB
LoadingTimestamp	timestamp	The timestamp loading began
ActiveTimestamp	timestamp	The timestamp when the vdb became active.

SYS.spatial_sys_ref

See also the PostGIS Documentation

Column name	Type	Description
srid	integer	Spatial Reference Identifier
auth_name	string	Name of the standard or standards body
auth_srid	integer	SRID for the auth_name authority
srtext	string	Well-Known Text representation
proj4text	string	For use with the Proj4 library

SYS.GEOMETRY_COLUMNS

See also the PostGIS Documentation

Column name	Type	Description
F_TABLE_CATALOG	string	catalog name
F_TABLE_SCHEMA	string	schema name
F_TABLE_NAME	string	table name
F_GEOMETRY_COLUMN	string	column name
COORD_DIMENSION	integer	Number of coordinate dimensions
SRID	integer	Spatial Reference Identifier
TYPE	string	Geometry type name

Note: The **coord_dimension** and **srid properties** are determined from the **{http://www.teiid.org/translator/spatial/2015}coord_dimension** and **{http://www.teiid.org/translator/spatial/2015}srid** extension properties on the column. When possible, these values are set automatically by the relevant importer. If the values are not set, they will be reported as **2** and **0**, respectively. If client logic expects actual values, such as integration with GeoServer, you can set these values manually.

SYS.ArrayIterate

Returns a resultset with a single column with a row for each value in the array.

```
SYS.ArrayIterate(IN val object[]) RETURNS TABLE (col object)
```

Example: ArrayIterate

```
select array_get(cast(x.col as string[]), 2) from (exec arrayiterate((('a', 'b'),('c','d')))) x
```

This will produce two rows - 'b', and 'd'.

8.2. SYSADMIN SCHEMA

System schema for administrative information and actions.

SYSADMIN.Usage

The following table supplies information about how views and procedures are defined.

Column name	Type	Description
VDBName	string	VDB name
UID	string	Object UID
object_type	string	Type of object (StoredProcedure, ForeignProcedure, Table, View, Column, etc.)
Name	string	Object Name or parent name
ElementName	string	Name of column or parameter, may be null to indicate a table/procedure. Parameter level dependencies are currently not implemented.
Uses_UID	string	Used object UID
Uses_object_type	string	Used object type
Uses_SchemaName	string	Used object schema
Uses_Name	string	Used object name or parent name
Uses_ElementName	string	Used column or parameter name, may be null to indicate a table/procedure level dependency

Every column, parameter, table, or procedure referenced in a procedure or view definition will be shown as used. Likewise every column, parameter, table, or procedure referenced in the expression that defines a view column will be shown as used by that column. No dependency information is shown for procedure parameters. Column level dependencies are not yet inferred through intervening temporary or common tables.

Example: SYSADMIN.Usage

```
SELECT * FROM SYSADMIN.Usage
```

Recursive common table queries can be used to determine transitive relationships.

Example: Finding all incoming usage

```
with im_using as (
   select 0 as level, uid, Uses_UID, Uses_Name, Uses_Object_Type, Uses_ElementName
     from usage where uid = (select uid from sys.tables where name='table name' and
schemaName='schema name')
   union all
   select level + 1, usage.uid, usage.Uses_UID, usage.Uses_Name, usage.Uses_Object_Type,
usage.Uses_ElementName
     from usage, im_using where level < 10 and usage.uid = im_using.Uses_UID) select * from
im_using
```

Example: Finding all outgoing usage

```
with uses_me as (
   select 0 as level, uid, Uses_UID, Name, Object_Type, ElementName
     from usage where uses_uid = (select uid from sys.tables where name='table name' and
schemaName='schema name')
   union all
   select level + 1, usage.uid, usage.Uses_UID, usage.Name, usage.Object_Type,
usage.ElementName
     from usage, uses_me where level < 10 and usage.uses_uid = uses_me.UID) select * from
uses_me
```

SYSADMIN.MatViews

The following table supplies information about all the materailized views in the virtual database.

Column name	Type	Description
VDBName	string	VDB name
SchemaName	string	Schema Name
Name	string	Short group name
TargetSchemaName	string	Name of the materialized table schema. Will be null for internal materialization.
TargetName	string	Name of the materialized table
Valid	boolean	True if materialized table is currently valid. Will be null for external materialization.

Column name	Type	Description
LoadState	boolean	The load state, can be one of **NEEDS_LOADING**, **LOADING**, **LOADED**, **FAILED_LOAD**. Will be null for external materialization.
Updated	timestamp	The timestamp of the last full refresh. Will be null for external materialization.
Cardinality	integer	The number of rows in the materialized view table. Will be null for external materialization.

Valid, LoadState, Updated, and Cardinality may be checked for external materialized views with the **SYSADMIN.matViewStatus** procedure.

Example: SYSADMIN.MatViews

```
SELECT * FROM SYSADMIN.MatViews
```

SYSADMIN.VDBResources

The following table provides the current VDB contents.

Column Name	Type	Description
resourcePath	string	The path to the contents.
contents	blob	The contents as a blob.

Example: SYSADMIN.VDBResources

```
SELECT * FROM SYSADMIN.VDBResources
```

SYSADMIN.Triggers

The following table provides the triggers in the virtual database.

Column name	Type	Description
VDBName	string	VDB name
SchemaName	string	Schema Name

Column name	Type	Description
TableName	string	Table name
Name	string	Trigger name
TriggerType	string	Trigger Type
TriggerEvent	string	Triggering Event
Status	string	Is Enabled
Body	clob	Trigger Action (FOR EACH ROW ...)
TableUID	string	Table Unique ID

Example: SYSADMIN.Triggers

```
SELECT * FROM SYSADMIN.Triggers
```

SYSADMIN.Views

The following table provides the views in the virtual database.

Column name	Type	Description
VDBName	string	VDB name
SchemaName	string	Schema Name
Name	string	View name
Body	clob	View Definition Body (SELECT ...)
UID	string	Table Unique ID

Example: SYSADMIN.Views

```
SELECT * FROM SYSADMIN.Views
```

SYSADMIN.StoredProcedures

The following table provides the StoredProcedures in the virtual database.

Column name	Type	Description
VDBName	string	VDB name
SchemaName	string	Schema Name
Name	string	Procedure name
Body	clob	Procedure Definition Body (BEGIN …)
UID	string	Unique ID

Example: SYSADMIN.StoredProcedures

```
SELECT * FROM SYSADMIN.StoredProcedures
```

SYSADMIN.Requests

The following table provides active requests against the virtual database.

```
VDBName string(255) NOT NULL,
```

Column name	Type	Description
VDBName	string	VDB name
SessionId	string	session identifier
ExecutionId	long	execution identifier
Command	clob	The query being executed
StartTimestamp	timestamp	Start timestamp
TransactionId	string	transaction identifier as reported by the Transaction Manager
ProcessingState	string	processing state, can be one of PROCESSING, DONE, CANCELED
ThreadState	string	thread state, can be one of RUNNING, QUEUED, IDLE

SYSADMIN.Sessions

The following table provides the Sessions active for the virtual database.

Column name	Type	Description
VDBName	string	VDB name
SessionId	string	session identifier
UserName	string	username
CreatedTime	timestamp	timestamp of when the session was created
ApplicationName	string	application name as reported by the client
IPAddress	string	IP Address as reported by the client

SYSADMIN.Transactions

The following table provides the active Transactions.

Column name	Type	Description
TransactionId	string	transaction identifier as reported by the Transaction Manager
SessionId	string	session identifier if a session is currently associated with the transaction
StartTimestamp	timestamp	start time of the transaction
Scope	string	scope of the transaction, can be one of GLOBAL, LOCAL, REQUEST, INHERITED. INHERITED means that a Transaction was already associated with the calling thread (embedded usage).

Note: Transactions that are not associated with a given session will always be shown. Transactions that are associated with a session must be for a session with the current VDB.

SYSADMIN.isLoggable

Tests if logging is enabled at the given level and context.

```
SYSADMIN.isLoggable(OUT loggable boolean NOT NULL RESULT, IN level string NOT NULL
DEFAULT 'DEBUG', IN context string NOT NULL DEFAULT 'org.teiid.PROCESSOR')
```

Returns true if logging is enabled. level can be one of the log4j levels: OFF, FATAL, ERROR, WARN, INFO, DEBUG, TRACE. level defaults to 'DEBUG' and context defaults to 'org.teiid.PROCESSOR'

Example: isLoggable

```
IF ((CALL SYSADMIN.isLoggable(context=>'org.something'))
BEGIN
  DECLARE STRING msg;
  // logic to build the message ...
  CALL SYSADMIN.logMsg(msg=>msg, context=>'org.something')
END
```

SYSADMIN.logMsg

Log a message to the underlying logging system.

```
SYSADMIN.logMsg(OUT logged boolean NOT NULL RESULT, IN level string NOT NULL DEFAULT
'DEBUG', IN context string NOT NULL DEFAULT 'org.teiid.PROCESSOR', IN msg object)
```

Returns true if the message was logged. level can be one of the log4j levels: OFF, FATAL, ERROR, WARN, INFO, DEBUG, TRACE. The level defaults to 'DEBUG' and context defaults to 'org.teiid.PROCESSOR'. A null msg object will be logged as the string 'null'.

Example: logMsg

```
CALL SYSADMIN.logMsg(msg=>'some debug', context=>'org.something')
```

The preceding example will log the message 'some debug' at the default level DEBUG to the context org.something.

8.2.1. SYSADMIN.refreshMatView

Full refresh/load of an internal materialized view. Returns integer RowsUpdated. -1 indicates a load is in progress, otherwise the cardinality of the table is returned. See the Caching Guide for more information.

See also SYSADMIN.loadMatView

```
SYSADMIN.refreshMatView(OUT RowsUpdated integer NOT NULL RESULT, IN ViewName string
NOT NULL, IN Invalidate boolean NOT NULL DEFAULT 'false')
```

8.2.2. SYSADMIN.refreshMatViewRow

Refreshes a row in an internal materialized view.

Returns integer RowsUpdated. -1 indicates the materialized table is currently invalid. 0 indicates that the specified row did not exist in the live data query or in the materialized table. See the Caching Guide for more information.

```
SYSADMIN.CREATE FOREIGN PROCEDURE refreshMatViewRow(OUT RowsUpdated integer
NOT NULL RESULT, IN ViewName string NOT NULL, IN Key object NOT NULL, VARIADIC
KeyOther object)
```

Example: SYSADMIN.refreshMatViewRow

The materialized view **SAMPLEMATVIEW** has 3 rows under the **TestMat** Model as below:

```
+-----+----+----+----+
| id  | a  | b  | c  |
+-----+----+----+----+
| 100 | a0 | b0 | c0 |
| 101 | a1 | b1 | c1 |
| 102 | a2 | b2 | c2 |
+-----+----+----+----+
```

Assuming the primary key only contains one column, id, update the second row:

```
EXEC SYSADMIN.refreshMatViewRow('TestMat.SAMPLEMATVIEW', '101')
```

Assuming the primary key contains more columns, a and b, update the second row:

```
EXEC SYSADMIN.refreshMatViewRow('TestMat.SAMPLEMATVIEW', '101', 'a1', 'b1')
```

8.2.3. SYSADMIN.refreshMatViewRows

Refreshes rows in an internal materialized view.

Returns integer RowsUpdated. -1 indicates the materialized table is currently invalid. Any row that does not exist in the live data query or in the materialized table will not count toward the RowsUpdated. For more information, see the Teiid Caching Guide.

```
SYSADMIN.refreshMatViewRows(OUT RowsUpdated integer NOT NULL RESULT, IN ViewName
string NOT NULL, VARIADIC Key object[] NOT NULL)
```

Example: SYSADMIN.refreshMatViewRows

Continuing use the **SAMPLEMATVIEW** in Example of SYSADMIN.refreshMatViewRow. Assuming the primary key only contains one column, id, update all rows:

```
EXEC SYSADMIN.refreshMatViewRows('TestMat.SAMPLEMATVIEW', ('100',), ('101',), ('102',))
```

Assuming the primary key comtain more columns, id, a and b compose of the primary key, update all rows:

```
EXEC SYSADMIN.refreshMatViewRows('TestMat.SAMPLEMATVIEW', ('100', 'a0', 'b0'), ('101', 'a1',
'b1'), ('102', 'a2', 'b2'))
```

8.2.4. SYSADMIN.setColumnStats

Set statistics for the given column.

```
SYSADMIN.setColumnStats(IN tableName string NOT NULL, IN columnName string NOT NULL, IN
distinctCount long, IN nullCount long, IN max string, IN min string)
```

All stat values are nullable. Passing a null stat value will leave corresponding metadata value unchanged.

8.2.5. SYSADMIN.setProperty

Set an extension metadata property for the given record. Extension metadata is typically used by Translators.

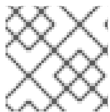
```
SYSADMIN.setProperty(OUT OldValue clob NOT NULL RESULT, IN UID string NOT NULL, IN Name
string NOT NULL, IN "Value" clob)
```

Setting a value to null will remove the property.

Example: Property Set

```
CALL SYSADMIN.setProperty(uid=>(SELECT uid FROM TABLES WHERE name='tab'),
name=>'some name', value=>'some value')
```

The preceding example will set the property 'some name'='some value' on table tab.



NOTE

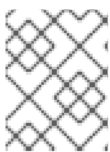
The use of this procedure will not trigger replanning of associated prepared plans.

Properties from built-in teiid_* namespaces can be set using the the short form - namespace:key form.

8.2.6. SYSADMIN.setTableStats

Set statistics for the given table.

```
SYSADMIN.setTableStats(IN tableName string NOT NULL, IN cardinality long NOT NULL)
```



NOTE

SYSADMIN.setColumnStats, SYSADMIN.setProperty, SYSADMIN.setTableStats are Metadata Procedures.

SYSADMIN.matViewStatus

matViewStatus is used to retrieve the status of materialized views via schemaName and viewName.

Returns tables which contains TargetSchemaName, TargetName, Valid, LoadState, Updated, Cardinality, LoadNumber, OnErrorAction.

```
SYSADMIN.matViewStatus(IN schemaName string NOT NULL, IN viewName string NOT NULL)
RETURNS TABLE (TargetSchemaName varchar(50), TargetName varchar(50), Valid boolean,
LoadState varchar(25), Updated timestamp, Cardinality long, LoadNumber long, OnErrorAction
varchar(25))
```

SYSADMIN.loadMatView

loadMatView is used to perform a complete refresh of an internal or external materialized table.

Returns integer RowsInserted. -1 indicates the materialized table is currently loading. And -3 indicates there was an exception when performing the load. See the Caching Guide for more information.

```
SYSADMIN.loadMatView(IN schemaName string NOT NULL, IN viewName string NOT NULL, IN
invalidate boolean NOT NULL DEFAULT 'false') RETURNS integer
```

Example: loadMatView

```
exec SYSADMIN.loadMatView(schemaName=>'TestMat',viewname=>'SAMPLEMATVIEW',
invalidate=>'true')
```

SYSADMIN.updateMatView

The updateMatView procedure is used to update a subset of an internal or external materialized table based on the refresh criteria.

The refresh criteria might reference the view columns by qualified name, but all instances of **.** in the view name will be replaced by _, because an alias is actually being used.

Returns integer RowsUpdated. -1 indicates the materialized table is currently invalid. And-3 indicates there was an exception when performing the update. See the Caching Guide for more information.

```
SYSADMIN.updateMatView(IN schemaName string NOT NULL, IN viewName string NOT NULL, IN
refreshCriteria string) RETURNS integer
```

SYSADMIN.updateMatView

Continuing use the **SAMPLEMATVIEW** in Example of SYSADMIN.refreshMatViewRow. Update view rows:

```
EXEC SYSADMIN.updateMatView('TestMat', 'SAMPLEMATVIEW', 'id = ''101'' AND a = ''a1''')
```

SYSADMIN.cancelRequest

Cancel the user request identified by execution id for the given session.

See also SYSADMIN.REQUESTS

```
SYSADMIN.cancelRequest(OUT cancelled boolean NOT NULL RESULT, IN SessionId string NOT
NULL, IN executionId long NOT NULL)
```

Example: Cancel

```
CALL SYSADMIN.cancelRequest('session id', 1)
```

SYSADMIN.terminateSession

Terminate the session with the given identifier.

See also SYSADMIN.SESSIONS

```
SYSADMIN.terminateSession(OUT terminated boolean NOT NULL RESULT, IN SessionId string
NOT NULL)
```

Example: Termination

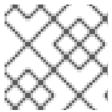
```
CALL SYSADMIN.terminateSession('session id')
```

SYSADMIN.terminateTransaction

Terminate the transaction associated with a session by marking the transaction as rollback only.

See also SYSADMIN.TRANSACTIONS

```
SYSADMIN.terminateTransaction(IN sessionid string NOT NULL)
```



NOTE

You cannot only cancel transactions that are associated with a session.

Example: Terminate

```
CALL SYSADMIN.terminateTransaction('session id')
```

CHAPTER 9. TRANSLATORS

Data Virtualization uses the Teiid Connector Architecture (TCA), which provides a robust mechanism for integrating with external systems. The TCA defines a common client interface between Data Virtualization and an external system that includes metadata as to what SQL constructs are available for pushdown and the ability to import metadata from the external system.

A Translator is the heart of the TCA and acts as the bridge logic between Data Virtualization and an external system.

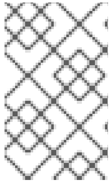
Translators can have a number of configurable properties. These are broken down into execution properties, which determine aspects of how data is retrieved, and import settings, which determine what metadata is read for import.

The execution properties for a translator typically have reasonable defaults. For specific translator types, such as the Derby translator, base execution properties are already tuned to match the source. In most cases the user will not need to adjust their values.

Table 9.1. Base execution properties - shared by all translators

Name	Description	Default
Immutable	Set to **true** to indicate that the source never changes. The transactional capability is reported as NONE, and update commands will fail.	false
RequiresCriteria	Set to **true** to indicate that source SELECT/UPDATE/DELETE queries require a where clause.	false
SupportsOrderBy	Set to **true** to indicate that the ORDER BY clause can be used.	false
SupportsOuterJoins	Set to **true** to indicate that OUTER JOINs can be used.	false
SupportsFullOuterJoins	If **SupportsOuterJoins** is set to **true**, **true** indicates that FULL OUTER JOINs can be used.	false
SupportsInnerJoins	Set to **true** to indicate that INNER JOINs can be used.	false
SupportedJoinCriteria	If join capabilities are enabled, defines the criteria that can be used as the join criteria. May be one of (ANY, THETA, EQUI, or KEY).	ANY

Name	Description	Default
MaxInCriteriaSize	If the use of **IN** criteria is enabled, specifies the maximum number of **IN** entries per predicate. **-1** indicates no limit.	-1
MaxDependentInPredicates	If the use of **IN** criteria is enabled, defines what the maximum number of predicates that can be used for a dependent join. Values less than 1 indicate to use only one **IN** predicate per dependent value pushed (which matches the pre-7.4 behavior).	-1
DirectQueryProcedureName	If **SupportsDirectQueryProcedure** is set to **true** for the translator, this property indicates the name of the procedure.	native
SupportsDirectQueryProcedure	Set to **true** to indicate that direct execution of commands is available for the translator.	false
ThreadBound	Set to **true** to indicate the translator's Executions should be processed by only a single thread	false
CopyLobs	If **true**, then returned large object (LOB) data (clob, blob, sql/xml) is copied by the engine in a memory safe manner. Use this option if the source does not provide memory safe LOBS or if you want to disconnect LOBS from the source connection.	false
TransactionSupport	The highest level of transaction capability. Used by the engine as a hint to determine if a transaction is needed for **autoCommitTxn=DETECT** mode. Can be one of XA, NONE, or LOCAL. If XA or LOCAL then access under a transaction will be serialized.	XA



NOTE

Only a subset of the available metadata can be set through execution properties on the base ExecutionFactory. All methods are available on the BaseDelegatingExecutionFactory.

There are no base importer settings.

Override execution properties

For all translators, you can override Execution Properties in the main vdb file.

Example: Overriding a translator property

.

```
CREATE FOREIGN DATA WRAPPER "oracle-override" TYPE oracle OPTIONS (RequiresCriteria 'true');

CREATE SERVER ora FOREIGN DATA WRAPPER "oracle-override" OPTIONS ("resource-name" 'java:/oracle');

CREATE SCHEMA ora SERVER ora;

SET SCHEMA ora;

IMPORT FROM SERVER ora INTO ora;
```

The preceding example overrides the *oracle* translator and sets the *RequiresCriteria* property to true. The modified translator is only available in the scope of this VDB. As many properties as desired may be overriden together.

See also VDB Definition.

Parameterizable native queries

In some situations the **teiid_rel:native-query property and native procedures accept parameterizable strings that can positionally reference IN parameters. A parameter reference has the form `$integer**, for example, **$1**. Note that one-based indexing is used and that only IN parameters may be referenced. Dollar-sign integer is therefore reserved, but may be escaped with another **$**`, for example, **$$1**. The value will be bound as a prepared value or a literal is a source specific manner. The native query must return a result set that matches the expectation of the calling procedure.

For example the native-query **select c from g where c1 = $1 and c2 = '$$1'** results in a JDBC source query of **select c from g where c1 = ? and c2 = '$1'**, where **?**` will be replaced with the actual value bound to parameter 1.

General import properties

Several import properties are shared by all translators.

When specifying an importer property, it must be prefixed with **importer.**. For example, **importer.tableTypes**.

Name	Description	Default
autoCorrectColumnNames	Replace any usage of **.** in a column name with _ as the period character is not valid in Data Virtualization column names.	true
renameDuplicateColumns	If true, rename duplicate columns caused by either mixed case collisions or **autoCorrectColumnNames** replacing **.** with _. A suffix **_n** where **n** is an integer will be added to make the name unique.	false
renameDuplicateTables	If true, rename duplicate tables caused by mixed case collisions. A suffix **_n** where **n** is an integer will be added to make the name unique.	false
renameAllDuplicates	If true, rename all duplicate tables, columns, procedures, and parameters caused by mixed case collisions. A suffix **_n** where **n** is an integer will be added to make the name unique. Supersedes the individual rename duplicate options.	false
nameFormat	Set to a Java string format to modify table and procedure names on import. The only argument will be the original name Data Virtualization name. For example use **prod_%s** to prefix all names with **prod_**.	

9.1. AMAZON S3 TRANSLATOR

The Amazon Simple Storage Service (S3) translator, known by the type name *amazon-s3*, exposes stored procedures to leverage Amazon S3 object resources.

This translator is typically used with the **TEXTTABLE** or **XMLTABLE** functions to consume CSV or XML formatted data, or to read Microsoft Excel files or other object files that are stored in Amazon S3. The S3 translator can access Amazon S3 by using an AWS access key ID and secret access key.

Usage

In the following example, a virtual database reads a CSV file with the name **g2.txt** from an Amazon S3 bucket called **teiidbucket**:

```
e1,e2,e3
5,'five',5.0
6,'six',6.0
7,'seven',7.0
```

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<vdb name="example" version="1">
    <model name="s3">
        <source name="web-connector" translator-name="user-s3" connection-jndi-
name="java:/amazon-s3"/>
    </model>
    <model name="Stocks" type="VIRTUAL">
        <metadata type="DDL"><![CDATA[
        CREATE VIEW G2 (e1 integer, e2 string, e3 double,PRIMARY KEY (e1))
            AS  SELECT SP.e1, SP.e2,SP.e3
                FROM (EXEC s3.getTextFile(name=>'g2.txt')) AS f,
                TEXTTABLE(f.file COLUMNS e1 integer, e2 string, e3 double HEADER) AS SP;
        ]]> </metadata>
    </model>
    <translator name="user-s3" type="amazon-s3">
      <property name="accesskey" value="xxxx"/>
      <property name="secretkey" value="xxxx"/>
      <property name="region" value="us-east-1"/>
      <property name="bucket" value="teiidbucket"/>
    </translator>
</vdb>
```

Execution properties

Use the translator override mechanism to supply the following properties.

Name	Description	Default
Encoding	The encoding that should be used for CLOBs returned by the getTextFiles procedure. The value should match an encoding known to the JRE.	The system default encoding.
Accesskey	Amazon security access key. Log in to Amazon console to find your security access key. When provided, this becomes the default access key.	n/a
Secretkey	Amazon security secret key. Log in to Amazon console to find your security secret key. When provided, this becomes the default secret key.	n/a

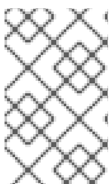
Name	Description	Default
Region	Amazon region to be used with the request. When provided, this will be default region used.	n/a
Bucket	Amazon S3 bucket name. If provided, this will serve as default bucket to be used for all the requests	n/a
Encryption	When server-side encryption with customer-provided encryption keys (SSE-C) is used, the key is used to define the "type" of encryption algorithm used. You can configure the translator to use the AES-256 or AWS-KMS encryption algorithms. If provided, this will be used as default algorithm for all "get" based calls.	n/a
Encryptionkey	When SSE-C type encryption used, where customer supplies the encryption key, this key will be used for defining the "encryption key". If provided, this will be used as default key for all "get" based calls.	n/a

TIP

For information about setting properties, see *Override execution property* in Translators, and review the examples in the sections that follow.

Procedures exposed by translator

When you add the a model (schema) like above in the example, the following procedure calls are available for user to execute against Amazon S3.



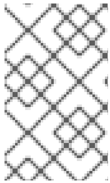
NOTE

bucket, **region**, **accesskey**, **secretkey**, **encryption** and **encryptionkey** are optional or nullable parameters in most of the methods provided. Provide them only if they are not already configured by using translator override properties as shown in preceding example.

getTextFile(...)

Retrieves the given named object as a text file from the specified bucket and region by using the provided security credentials as CLOB.

```
getTextFile(string name NOT NULL, string bucket, string region,
  string endpoint, string accesskey, string secretkey,string encryption, string encryptionkey, boolean
stream default false)
  returns TABLE(file blob, endpoint string, lastModified string, etag string, size long);
```



NOTE

endpoint is optional. When provided the endpoint URL is used instead of the one constructed by the supplied properties. Use **encryption** and **encryptionkey** only in when server side security with customer supplied keys (SSE-C) in force.

If the value of **stream** is true, then returned LOBs are read only once and are not typically buffered to disk.

Examples

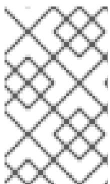
```
exec getTextFile(name=>'myfile.txt');

SELECT SP.e1, SP.e2,SP.e3, f.lastmodified
  FROM (EXEC getTextFile(name=>'myfile.txt')) AS f,
  TEXTTABLE(f.file COLUMNS e1 integer, e2 string, e3 double HEADER) AS SP;
```

getFile(...)

Retrieves the given named object as binary file from specified bucket and region using the provided security credentials as BLOB.

```
getFile(string name NOT NULL, string bucket, string region,
  string endpoint, string accesskey, string secretkey, string encryption, string encryptionkey, boolean
stream default false)
  returns TABLE(file blob, endpoint string, lastModified string, etag string, size long)
```



NOTE

endpoint is optional. When provided the endpoint URL is used instead of the one constructed by the supplied properties. Use **encryption** and **encryptionkey** only in when server side security with customer supplied keys (SSE-C) in force.

If the value of **stream** is true, then returned lOBs are read once and are not typically buffered to disk.

Examples

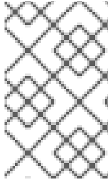
```
exec getFile(name=>'myfile.xml', bucket=>'mybucket', region=>'us-east-1', accesskey=>'xxxx',
secretkey=>'xxxx');

select b.* from (exec getFile(name=>'myfile.xml', bucket=>'mybucket', region=>'us-east-1',
accesskey=>'xxxx', secretkey=>'xxxx')) as a,
XMLTABLE('/contents' PASSING XMLPARSE(CONTENT a.result WELLFORMED) COLUMNS e1
integer, e2 string, e3 double) as b;
```

saveFile(...)

Save the CLOB, BLOB, or XML value to given name and bucket. In the following procedure signature, the *contents* parameter can be any of the LOB types.

```
call saveFile(string name NOT NULL, string bucket, string region, string endpoint,
  string accesskey, string secretkey, contents object)
```



NOTE

You cannot use **saveFile** to stream or chunk uploads of a file's contents. If you try to load very large objects, out-of-memory issues can result. You cannot configure **saveFile** to use SSE-C encryption.

Examples

```
exec saveFile(name=>'g4.txt', contents=>'e1,e2,e3\n1,one,1.0\n2,two,2.0');
```

deleteFile(...)

Delete the named object from the bucket.

```
call deleteFile(string name NOT NULL, string bucket, string region, string endpoint, string accesskey,
string secretkey)
```

Examples

```
exec deleteFile(name=>'myfile.txt');
```

list(...)

Lists the contents of the bucket.

```
call list(string bucket, string region, string accesskey, string secretkey, nexttoken string)
   returns Table(result clob)
```

The result is the XML file that Amazon S3 provides in the following format

```
<?xml version="1.0" encoding="UTF-8"?>/n
<ListBucketResult
   xmlns="http://s3.amazonaws.com/doc/2006-03-01/">
   <Name>teiidbucket</Name>
   <Prefix></Prefix>
   <KeyCount>1</KeyCount>
   <MaxKeys>1000</MaxKeys>
   <IsTruncated>false</IsTruncated>
   <Contents>
      <Key>g2.txt</Key>
      <LastModified>2017-08-08T16:53:19.000Z</LastModified>
      <ETag>&quot;fa44a7893b1735905bfcce59d9d9ae2e&quot;</ETag>
      <Size>48</Size>
```

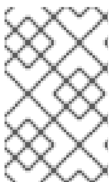
```
      <StorageClass>STANDARD</StorageClass>
    </Contents>
</ListBucketResult>
```

You can parse this into a view by using a query similar to the one in the following example:

```
select b.* from (exec list(bucket=>'mybucket', region=>'us-east-1')) as a,
 XMLTABLE(XMLNAMESPACES(DEFAULT 'http://s3.amazonaws.com/doc/2006-03-01/'),
'/ListBucketResult/Contents'
 PASSING XMLPARSE(CONTENT a.result WELLFORMED) COLUMNS Key string, LastModified
string, ETag string, Size string,
 StorageClass string, NextContinuationToken string PATH '../NextContinuationToken') as b;
```

If all properties (**bucket**, **region**, **accesskey**, and **secretkey**) are defined as translator override properties, you can run the following simple query:

```
SELECT * FROM Bucket
```



NOTE

If there are more then 1000 object in the bucket, then the value 'NextContinuationToken' need to be supplied as 'nexttoken' into the *list* call to fetch the next batch of objects. This can be automated in Data Virtualization with enhancement request.

9.2. DELEGATOR TRANSLATORS

You can use the *delegator* translator, which is available in the core Data Virtualization installation, to modify the capabilities of a existing translator. Often times for debugging purposes, or in special situations, you might want to turn certain capabilities of a translator on or off. For example, say that the latest version of a Hive database supports the **ORDER BY** construct, but the current Data Virtualization version of the Hive translator does not. You could use the delegator translator to enable **ORDER BY** compatibility without actually writing any code. Similarly, you could do the reverse, and turn off certain capabilities to produce a better plan.

To use the delegator translator, you must define it in the DDL. The following example shows how to override the "hive" translator and turn off the **ORDER BY** capability.

```
CREATE DATABASE myvdb;
USE DATABASE myvdb;
CREATE FOREIGN DATA WRAPPER "hive-delegator" TYPE delegator OPTIONS (delegateName
'hive', supportsOrderBy 'false');
CREATE SERVER source FOREIGN DATA WRAPPER "hive-delegator" OPTIONS ("resource-
name" 'java:hive-ds');
CREATE SCHEMA mymodel SERVER source;
SET SCHEMA mymodel;
IMPORT FROM SERVER source INTO mymodel;
```

For more information about the translator capabilities that you can override by using execution properties, see *Translator_Capabilities* in the Translator Development Guide. The preceding example shows how you might modify the default **ORDER BY** compatibility of the Hive translator.

9.2.1. Extending the delegator translator

You can create a delegating translator by extending the **org.teiid.translator.BaseDelegatingExecutionFactory**. After your classes are packaged as a custom translator, you can wire another translator instance into your delegating translator at runtime in order to intercept all of the calls to the delegate. This base class does not provide any functionality on its own, other than delegation. You can hard code capabilities into the translator instead of defining them as part of the DDL configuration. You can also override methods to provide alternate behavior.

Table 9.2. Execution properties

Name	Description	Default
delegateName	Translator instance name to delegate to.	n/a
cachePattern	Regex pattern of queries that should be cached using the translator caching API.	n/a
cacheTtl	Time to live in milliseconds for queries matching the cache pattern.	n/a

For example, if you use the *oracle* translator in your virtual database, and you want to intercept calls that go through the translator, you could write a custom delegating translator, as in the following example:

```
@Translator(name="interceptor", description="interceptor")
public class InterceptorExecutionFactory extends
org.teiid.translator.BaseDelegatingExecutionFactory{
   @Override
   public void getMetadata(MetadataFactory metadataFactory, C conn) throws TranslatorException {
      // do intercepting code here..

      // If you want call the original delegate, do not call if do not need to.
      // but if you did not call the delegate fullfill the method contract
      super.getMetadata(metadataFactory, conn);

      // do more intercepting code here..
   }
}
```

You could then deploy this translator in the Data Virtualization engine. Then in your DDL file, define an interceptor translator as in the following example:

```
CREATE DATABASE myvdb VERSION '1';
USE DATABASE myvdb VERSION '1';
CREATE FOREIGN DATA WRAPPER "oracle-interceptor" TYPE interceptor OPTIONS
(delegateName 'oracle');
CREATE SERVER source FOREIGN DATA WRAPPER "oracle-interceptor" OPTIONS ("resource-
name" 'java:oracle-ds');
```

```
CREATE SCHEMA mymodel SERVER source;
SET SCHEMA mymodel;
IMPORT FROM SERVER source INTO mymodel;
```

We have defined a "translator" override called **oracle-interceptor**, which is based on the custom translator "interceptor" from above, and supplied the translator it needs to delegate to "oracle" as its delegateName. Then, we used this override translator **oracle-interceptor** in the VDB. Future calls going into this VDB model's translator are intercepted by your code to do whatever you want to do.

9.3. FILE TRANSLATOR

The file translator, known by the type name *file*, exposes stored procedures to leverage file resources. The translator is typically used with the **TEXTTABLE** or **XMLTABLE** functions to consume CSV or XML formatted data.

Table 9.3. Execution properties

Name	Description	Default
Encoding	The encoding that should be used for CLOBs returned by the getTextFiles procedure. The value should match an encoding known to Data Virtualization. For more information, see *TO_CHARS* and *TO_BYTES* in String functions.	The system default encoding.
ExceptionIfFileNotFound	Throw an exception in getFiles or getTextFiles if the specified file/directory does not exist.	true

TIP

For information about how to set properties, see the following example, and *Override execution properties* in Translators.

Example: Virtual datbase DDL override

```
CREATE SERVER "file-override"
   FOREIGN DATA WRAPPER file
   OPTIONS(
      Encoding 'ISO-8859-1', "ExceptionIfFileNotFound" false
   );

CREATE SCHEMA file SERVER "file-override";
```

getFiles

```
getFiles(String pathAndPattern) returns
TABLE(file blob, filePath string, lastModified timestamp, created timestamp, size long)
```

Retrieve all files as BLOBs matching the given path and pattern.

```
call getFiles('path/*.ext')
```

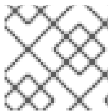
If the path is a directory, then all files in the directory are returned. If the path matches a single file, the file is returned.

The * character is treated as a wildcard to match any number of characters in the path name. Zero or matching files will be returned.

If ***' is not used, and if the path doesn't exist and `ExceptionIfFileNotFound** is true, then an exception is raised.

getTextFiles

```
getTextFiles(String pathAndPattern) returns
TABLE(file clob, filePath string, lastModified timestamp, created timestamp, size long)
```



NOTE

The size reports the number of bytes.

Retrieve all files as CLOBs matching the given path and pattern.

```
call getTextFiles('path/*.ext')
```

Retrieves the same files **getFiles**, but with the difference that the results are CLOB values that use the encoding execution property as the character set.

saveFile

Save the CLOB, BLOB, or XML value to given path

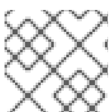
```
call saveFile('path', value)
```

deleteFile

Delete the file at the given path

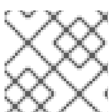
```
call deleteFile('path')
```

The path should reference an existing file. If the file does not exist and **ExceptionIfFileNotFound** is true, then an exception will be thrown. An exception is also thrown if the file cannot be deleted.



NATIVE QUERIES

This feature is not applicable for the File translator.



DIRECT QUERY PROCEDURE

This feature is not applicable for the File translator.

9.4. GOOGLE SPREADSHEET TRANSLATOR

The *google-spreadsheet* translator is used to connect to a Google Sheets spreadsheet.

The query approach expects that the data in the worksheet has the following characteristics:

- All columns that contains data can be queried.
- Any column with an empty cell has the value retrieved as null. However, differentiating between null string and empty string values may not always be possible as Google treats them interchangeably. Where possible, the translator may provide a warning or throw an exception if it cannot differentiate between null and empty strings.
- If the first row is present and contains string values, then the row is assumed to represent the column labels.

If you are using the default native metadata import, the metadata for your Google account (worksheets and information about columns in worksheets) is loaded upon translator start up. If you make any changes in data types, it is advisable to restart your virtual database.

The translator can submit queries against a single sheet only. It provides ordering, aggregation, basic predicates, and most of the functions available in the spreadsheet query language.

The google-spreadsheet translator does not provide importer settings, but it can provide metadata for VDBs.

WARNING

If you remove all data rows from a sheet with a header that is defined in Data Virtualization, you can no longer access the sheet through Data Virtualization. The Google API will treat the header as a data row at that point, and queries to it will no longer be valid.

WARNING

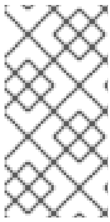
Non-string fields are updated using the canonical Data Virtualization SQL value. In cases where the spreadsheet is using a non-conforming locale, consider disallowing updates. For more information, see TEIID-4854 and the following information about the **allTypesUpdatable** import property.

Importer properties

- *allTypesUpdatable*- Set to true to mark all columns as updatable. Set to false to enable update only on string or Boolean columns that are not affected by TEIID-4854. Defaults to true.

Native queries

Google spreadsheet source procedures may be created using the **teiid_rel:native-query** extension. For more information, see *Parameterizable native queries* in Translators. The procedure will invoke the native-query similar to an native procedure call, with the benefits that the query is predetermined, and that result column types are known, rather than requiring the use of ARRAYTABLE or similar functionality. For more information, see the *Select* section that follows.



DIRECT QUERY PROCEDURE

This feature is turned off by default, because of the security risk in permitting any command to execute against the data source. To enable this feature, set the property *SupportsDirectQueryProcedure* to true. For more information, see *Override execution properties* in Translators.

TIP

By default the name of the procedure that executes the queries directly is called **native**. You can change its name by overriding the execution property *DirectQueryProcedureName*. For more information, see *Override execution properties* in Translators.

The Google spreadsheet translator provides a procedure to execute any ad-hoc query directly against the source without any Data Virtualization parsing or resolving. Because the metadata of this procedure's execution results are not known to Data Virtualization, they are returned as an object array. You can use ARRAYTABLE to construct tabular output for consumption by client applications. For more information, see ARRAYTABLE.

Data Virtualization exposes this procedure with a simple query structure as shown in the following example:

Select example

```
SELECT x.* FROM (call google_source.native('worksheet=People;query=SELECT A, B, C')) w,
 ARRAYTABLE(w.tuple COLUMNS "id" string , "type" string, "name" String) AS x
```

The first argument takes semicolon-separated (;) name-value pairs of the following properties to execute the procedure:

Property	Description	Required
worksheet	Google spreadsheet name.	yes
query	Spreadsheet query.	yes
limit	Number of rows to fetch.	no
offset	Offset of rows to fetch from limit or beginning.	no

9.5. JDBC TRANSLATORS

The JDBC translators bridge the SQL semantics and data type differences between Data Virtualization and a target RDBMS. Data Virtualization has a range of specific translators that target the most popular open source and proprietary relational databases.

Usage

Usage of a JDBC source is straight-forward. Using Data Virtualization SQL, the source can be queried as if the tables and procedures were local to the Data Virtualization system.

If you are using a relational data source, or a data source that has a JDBC driver, and you do not find a specific translator available for that data source type, then start with the JDBC ANSI translator. The JDBC ANSI translator should enable you to perform the SQL basics. If there specific data source capabilities that are not available, you can define a custom translator that does what you need. For more information, see Translator Development.

Type conventions

UID types including UUID, GUID, or UNIQUEIDENTIFIER are typically mapped to the Data Virtualization string type. JDBC data sources treat UID strings as non-case sensitive, but they are case-sensitive in Data Virtualization. If the source does not support the implicit conversion to the string type, then usage in functions that expect a string value might fail at the source.

The following table lists the execution properties that are shared by all JDBC translators.

Table 9.4. Execution properties – Shared by all JDBC translators

Name	Description	Default
DatabaseTimeZone	The time zone of the database. Used when fetching date, time, or timestamp values.	The system default time zone
DatabaseVersion	The specific database version. Used to further tune the use of pushdown operations.	The base compatible version, or the version that is derived from the DatabaseMetadata.getDatabaseProductVersion string. Automatic detection requires a connection. If there are circumstances where you are getting an exception due to capabilities being unavailable (for example, because a connection is not available), then set **DatabaseVersion** property. Use the **JDBCExecutionFactory.usesDatabaseVersion()**` method to control whether your translator requires a connection to determine capabilities.

Name	Description	Default
TrimStrings	**true** trims trailing whitespace from fixed length character strings. Note that Data Virtualization only has a string, or varchar, type that treats trailing whitespace as meaningful.	false
RemovePushdownCharacters	Set to a regular expression to remove characters that not allowed or undesirable for the source. For example **[\u0000]** will remove the null character, which is problematic for sources such as PostgreSQL and Oracle. Note that this does effectively change the meaning of the affected string literals and bind values, which must be carefully considered.	
UseBindVariables	**true** indicates that PreparedStatements should be used and that literal values in the source query should be replace with bind variables. If **false** only LOB values will trigger the use of PreparedStatements.	true
UseCommentsInSourceQuery	This will embed a leading comment with session/request id in the source SQL for informational purposes. Can be customized with the CommentFormat property.	false

Name	Description	Default
CommentFormat	MessageFormat string to be used if **UseCommentsInSourceQuery** is enabled. You can set the format to one of the following values: • 0 - Session ID string. • 1 - Parent request ID string. • 2 - Request part ID string. • 3 - Execution count ID string. • 4 - User name string. • 5 - VDB name string. • 6 - VDB version integer. • 7 - Is transactional boolean.	**/*teiid sessionid:\{0}, requestid:\{1}.\{2}*/**
MaxPreparedInsertBatchSize	The max size of a prepared insert batch.	2048
StructRetrieval	Specify one of the following Struct retrieval modes: • OBJECT - **getObject** value returned. • COPY - Returned as a **SerialStruct**. • ARRAY - Returned as an array.	OBJECT
EnableDependentJoins	Allow dependent join pushdown for sources that use temporary tables (DB2, Derby, H2, HSQL 2.0+, MySQL 5.0+, Oracle, PostgreSQL, SQLServer, SQP IQ, Sybase).	false

Importer properties – Shared by all JDBC translators

When specifying the importer property, it must be prefixed with **importer.**. Example: **importer.tableTypes**

Name	Description	Default
catalog	See DatabaseMetaData.getTables [1]	null
schemaName	Recommended setting to import from a single schema. The schema name will be converted into an escaped pattern,overriding schemaPattern if it is also set.	null
schemaPattern	See DatabaseMetaData.getTables [1]	null
tableNamePattern	See DatabaseMetaData.getTables [1]	null
procedureNamePattern	See DatabaseMetaData.getProcedures [1]	null
tableTypes	Comma separated list – without spaces – of imported table types. See **DatabaseMetaData.getTables** [1]	null
excludeTables	A case-insensitive regular expression that when matched against a fully qualified table name [2] will exclude it from import. Applied after table names are retrieved. Use a negative look-ahead (?!<inclusion pattern>).* to act as an inclusion filter.	null
excludeProcedures	A case-insensitive regular expression that when matched against a fully qualified procedure name [2] will exclude it from import. Applied after procedure names are retrieved. Use a negative look-ahead (?!<inclusion pattern>).* to act as an inclusion filter.	null

Name	Description	Default
importKeys	**true** to import primary and foreign keys. NOTE: Foreign keys to tables that are not imported will be ignored.	true
autoCreateUniqueConstraints	**true** to create a unique constraint if one is not found for a foreign keys	true
importIndexes	**true** to import index/unique key/cardinality information	false
importApproximateIndexes	**true** to import approximate index information. See DatabaseMetaData.getIndexInfo [1]. WARNING: Setting to **false** may cause lengthy import times.	true
importProcedures	**true** to import procedures and procedure columns - Note that it is not always possible to import procedure result set columns due to database limitations. It is also not currently possible to import overloaded procedures.	false
importSequences	**true** to import sequences. Compatible only with Db2, Oracle, PostgreSQL, SQL Server, and H2. A matching sequence will be imported to a 0-argument Data Virtualization function **name_nextval**.	false
sequenceNamePattern	LIKE pattern string to use when importing sequences. Null or **%** will match all.	null

Name	Description	Default
useFullSchemaName	When **false**, directs the importer to use only the object name as the Data Virtualization name. It is expected that all objects will come from the same foreign schema. When **true** (not recommended) the Data Virtualization name will be formed using the catalog and schema names as directed by the **useCatalogName** and **useQualifiedName** properties, and it will be allowed for objects to come from multiple foreign schema. This option does not affect the name in source property.	false (only change when importing from multiple foreign schema).
useQualifiedName	**true** will use name qualification for both the Data Virtualization name and name in source as further refined by the **useCatalogName** and **useFullSchemaName** properties. Set to **false** to disable all qualification for both the Data Virtualization name and the name in source, which effectively ignores the **useCatalogName** and **useFullSchemaName** properties. WARNING: When you set this option to **false**, it can lead to objects with duplicate names when importing from multiple schemas, which results in an exception.	true (rarely needs changed)

Name	Description	Default
useCatalogName	**true** will use any non-null/non-empty catalog name as part of the name in source, e.g. "catalog"."schema"."table"."column", and in the Data Virtualization runtime name if applicable. **false** will not use the catalog name in either the name in source nor the Data Virtualization runtime name. Only required to be set to **false** for sources such as HSQL that do not use the catalog concept, but return a non-null/non-empty catalog name in their metadata.	true (rarely needs changed)
widenUnsignedTypes	**true** to convert unsigned types to the next widest type. For example, SQL Server reports **tinyint** as an unsigned type. With this option enabled, **tinyint** would be imported as a short instead of a byte.	true
useIntegralTypes	**true** to use integral types rather than decimal when the scale is 0.	false
quoteNameInSource	**false** will override the default and direct Data Virtualization to create source queries using unquoted identifiers.	true
useAnyIndexCardinality	**true** will use the maximum cardinality returned from **DatabaseMetaData.getIndexInfo**. **importKeys** or **importIndexes** needs to be enabled for this setting to have an effect. This allows for better stats gathering from sources that don't return a statistical index.	false
importStatistics	**true** will use database dependent logic to determine the cardinality if none is determined. Not available for all database types – currently available for Oracle and MySQL only.	false

Name	Description	Default
importRowIdAsBinary	**true** will import **RowId** columns as varbinary values.	false
importLargeAsLob	**true** will import character and binary types larger than the Data Virtualization max as CLOB or BLOB respectively. If you experience memory issues even with the property enabled, you should use the **copyLob** execution property as well.	false

[1] JavaDoc for DatabaseMetaData
[2] The fully qualified name for exclusion is based upon the settings of the translator and the particulars of the database. All of the applicable name parts used by the translator settings (see **useQualifiedName** and **useCatalogName**) including catalog, schema, table will be combined as **catalogName.schemaName.tableName** with no quoting. For example, Oracle does not report a catalog, so the name used with default settings for comparison would be just **schemaName.tableName**.

WARNING

The default import settings will crawl all available metadata. This import process is time-consuming, and full metadata import is not needed in most situations. Most commonly you'll want to limit the import by at least **schemaName** or **schemaPattern** and **tableTypes**.

Example: Importer settings to import only tables and views from my-schema.

```
SET SCHEMA ora;

IMPORT FOREIGN SCHEMA "my-schema" FROM SERVER ora INTO ora OPTIONS
("importer.tableTypes" 'TABLE,VIEW');
```

For more information about importer settings, see Virtual databases.

Native queries

Physical tables, functions, and procedures may optionally have native queries associated with them. No validation of the native query is performed, it is simply used in a straight-forward manner to generate the source SQL. For a physical table setting the **teiid_rel:native-query** extension metadata will execute the native query as an inline view in the source query. This feature should only be used against sources that provide inline views. The native query is used as is and is not treated as a parameterized string. For example, on a physical table **y** with **nameInSource="x"`** and **teiid_rel:native-query="select c from g"**, the Data Virtualization source query **"SELECT c FROM y"** would generate the SQL query **"SELECT c FROM (select c from g) as x"**. Note that the column names in the native query must match the **nameInSource** of the physical table columns for the resulting SQL to be valid.

For physical procedures you may also set the **teiid_rel:native-query** extension metadata to a desired query string with the added ability to positionally reference IN parameters . For more information, see *Parameterizable native queries* in Translators. The **teiid_rel:non-prepared** extension metadata property can be set to **false** to turn off parameter binding.

Be careful in setting this option, because inbound allows for SQL injection attacks if not properly validated. The native query does not need to call a stored procedure. Any SQL that returns a result set that positionally matches the result set that is expected by the physical stored procedure metadata will work. For example on a stored procedure **x** with **teiid_rel:native-query="select c from g where c1 = $1 and c2 = `$$1"**, the Data Virtualization source query **`"CALL x(?)"`** would generate the SQL query **`"select c from g where c1 = ? and c2 = `$1"`**. Note that **?** in this example will be replaced with the actual value bound to parameter 1.

Direct query procedure

This feature is turned off by default, because of the inherent security risk in allowing any command to be run against the source. To enable this feature, override the execution property called *SupportsDirectQueryProcedure* and set it to **true**. For more information, see *Override execution properties* in Translators.

By default, the name of the procedure that executes the queries directly is **native**. To change the name, override the execution property *DirectQueryProcedureName*.

The JDBC translator provides a procedure to execute any ad-hoc SQL query directly against the source without Data Virtualization parsing or resolving. Since the metadata of this procedure's results are not known to Data Virtualization, they are returned as an object array. ARRAYTABLE can be used construct tabular output for consumption by client applications. For more information, see arraytable.

SELECT example

```
SELECT x.* FROM (call jdbc_source.native('select * from g1')) w,
 ARRAYTABLE(w.tuple COLUMNS "e1" integer , "e2" string) AS x
```

INSERT example

```
SELECT x.* FROM (call jdbc_source.native('insert into g1 (e1,e2) values (?, ?)', 112, 'foo')) w,
 ARRAYTABLE(w.tuple COLUMNS "update_count" integer) AS x
```

UPDATE example

```
SELECT x.* FROM (call jdbc_source.native('update g1 set e2=? where e1 = ?','blah', 112)) w,
 ARRAYTABLE(w.tuple COLUMNS "update_count" integer) AS x
```

DELETE example

```
SELECT x.* FROM (call jdbc_source.native('delete from g1 where e1 = ?', 112)) w,
 ARRAYTABLE(w.tuple COLUMNS "update_count" integer) AS x
```

9.5.1. Actian Vector translator (actian-vector)

Also see common JDBC Translators information.

The Actian Vector translator, known by the type name ***actian-vector***, is for use with Actian Vector in Hadoop.

Download the JDBC driver at http://esd.actian.com/platform. Note the port number in the connection URL is "AH7", which maps to 16967.

9.5.2. Apache Phoenix Translator (phoenix)

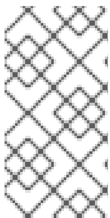
Also see common JDBC Translators information.

The Apache Phoenix translator, known by the type name ***phoenix***, exposes querying functionality to HBase tables. Apache Phoenix is a JDBC SQL interface for HBase that is required for this translator as it pushes down commands into Phoenix SQL.

The translator is also known by the deprecated name ***hbase***. The name change reflects that the translator is specific to Phoenix and that there could be other translators introduced in the future to connect to HBase.

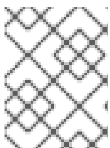
Do not use the **DatabaseTimezone** property with this translator.

The HBase translator cannot process Join commands. Phoenix uses the HBase Table Row ID as the Primary Key. This Translator is developed with Phoenix 4.3 or greater for HBase 0.98.1 or greater.



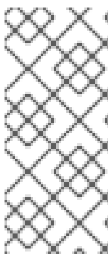
NOTE

The translator implements **INSERT**/**UPDATE** through the Phoenix **UPSERT** operation. This means you can see different behavior than with standard **INSERT**/**UPDATE**. For example, repeated inserts will not throw a duplicate key exception, but will instead update the row in question.



NOTE

Due to Phoenix driver limitations, the importer does not look for unique constraints, and defaults to not importing foreign keys.



NOTE

The translator can process SQL **OFFSET** arguments and other features starting with Phoenix 4.8. The Phoenix driver hard codes the server version in **PhoenixDatabaseMetaData**, and does not otherwise provide a way to detect the server version at runtime. If a newer driver is used with an older server, set the database version translator property manually.

WARNING

The Phoenix driver does not have robust handling of time values. If your time values are normalized to use a date component of 1970-01-01, then the default handling will work correctly. If not, then the time column should be modeled as timestamp instead.

9.5.3. Cloudera Impala translator (impala)

Also see common JDBC Translators information.

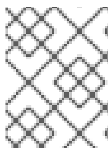
The Cloudera Impala translator, known by the type name ***impala***, is for use with Cloudera Impala 1.2.1 or later.

Impala has limited support for data types. It is does not have native support for time/date/xml or LOBs. These limitations are reflected in the translator capabilities. A Data Virtualization view can use these types, however the transformation would need to specify the necessary conversions. Note that in those situations, the evaluations will be done in the Data Virtualization engine.

Do not use the **DatabaseTimeZone** translator property with the Impala translator.

Impala only supports EQUI join, so using any other joins types on its source tables will result in inefficient queries.

To write criteria based on partitioned columns, model them on the source table, but do not include them in selection columns.



NOTE

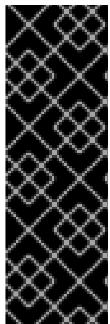
Impala Hive importer does not have concept of catalog or source schema, nor does it import keys, procedures, indexes, etc.

Impala specific importer properties

useDatabaseMetaData

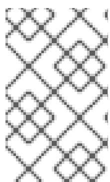
Set to true to use the normal import logic with the option to import index information disabled. Defaults to false.

If the value of **useDatabaseMetaData** is false, the typical JDBC **DatabaseMetaData** calls are not used, so not all of the common JDBC importer properties are applicable to Impala. You may still use **excludeTables**, regardless.



IMPORTANT

Some versions of Impala require the use of a **LIMIT** when performing an **ORDER BY**. If no default is configured in Impala, an exception can occur when a Data Virtualization query with an **ORDER BY** but no **LIMIT** is issued. You should set an Impala-wide default, or configure the connection pool to use a new connection SQL string to issue a **SET DEFAULT_ORDER_BY_LIMIT** statement. For more information about Impala limit options, such as how to control what happens when the limit is exceeded, see the Cloudera documentation.



NOTE

If the Impala JDBC driver has problems processing **PreparedStatements** or parsing statements in general, try disabling **useBindVariables**. For more information, see https://issues.redhat.com/browse/TEIID-4610.

9.5.4. Db2 Translator (db2)

Also see common JDBC Translators information.

The Db2 translator, known by the type name ***db2***, is for use with IBM Db2 V8 or later, or IBM Db2 for i V5.4 or later.

Db2 execution properties

Db2 execution properties

DB2ForI

Indicates that the the Db2 instance is Db2 for i. Defaults to **false**.

supportsCommonTableExpressions

Indicates that the Db2 instance supports common table expressions (CTEs). Defaults to **true**. Common table expression are not fully supported on some older versions of Db2, and on instances of Db2 that run in a conversion mode. If you encounter errors working with CTEs in these environments, set the CTE property to **false**.

9.5.5. Derby translator (derby)

Also see common JDBC Translators information.

The Derby translator, known by the type name ***derby***, is for use with Derby 10.1 or later.

9.5.6. Exasol translator (exasol)

Also see common JDBC Translators information.

The Exasol translator, known by the type name ***exasol***, is for use with Exasol version 6 or later.

Usage

The Exasol database has the NULL HIGH default ordering, whereas the Data Virtualization engine works in the NULL LOW mode. As a result, depending on whether the ordering is pushed down to Exasol or done by the engine, you might observe NULLs at either the beginning or end of returned results. To enforce consistency, you can run Data Virtualization with **org.teiid.pushdownDefaultNullOrder=true** to specify NULL LOW ordering. Enforcing NULL LOW ordering can result in decreased performance.

9.5.7. Greenplum Translator (greenplum)

Also see common JDBC Translators information.

The Greenplum translator, known by the type name ***greenplum***, is for use with the Greenplum database. This translator is an extension of the PostgreSQL translator, and inherits its options.

9.5.8. H2 Translator (h2)

Also see common JDBC Translators information.

The H2 Translator, known by the type name ***h2***, is for use with H2 version 1.1 or later.

9.5.9. Hive Translator (hive)

Also see common JDBC Translators information.

The Hive translator, known by the type name ***hive***, is for use with Hive v.10 and SparkSQL v1.0 and later.

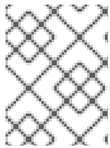
Capabilities

Hive is compatible with a limited set of data types. It does not have native support for time/XML or large objects (LOBs). These limitations are reflected in the translator capabilities. Although a Data Virtualization view can use these types, the transformation must specify the necessary conversions. Note that in those situations, evaluations are processed in Data Virtualization engine.

Do not use the **DatabaseTimeZone** translator property with the Hive translator.

Hive only supports EQUI join, so using any other joins types on its source tables will result in inefficient queries.

To write criteria based on partitioned columns, model them on the source table, but do not include them in selection columns.



NOTE

The Hive importer does not have the concept of catalog or source schema, nor does it import keys, procedures, indexes, and so forth.

Import properties

trimColumnNames

For Hive 0.11.0 and later, the **DESCRIBE** command metadata is inappropriately returned with padding. Set this property to **true** to remove white space from column names. Defaults to **false**.

useDatabaseMetaData

For Hive 0.13.0 and later, the normal JDBC **DatabaseMetaData** facilities are sufficient to perform an import. Set to **true** to use the normal import logic with the option to import index information disabled. Defaults to **false**. When true, **trimColumnNames** has no effect. If it is set to false, the typical JDBC DatabaseMetaData calls are not used, so not all of the common JDBC importer properties are applicable to Hive. You can still use excludeTables anyway.

"Database Name"

When the database name used in Hive differs from **default**, the metadata retrieval and execution of queries does not work as expected in Data Virtualization. The Hive JDBC driver seems to be implicitly connecting (tested with < 0.12) to "default" database, thus ignoring the database name mentioned on connection URL. You can work around this issue if you configure your connection source to send the command **use {database-name}**.

This is fixed in version 0.13 and later of the Hive JDBC driver. For more information, see https://issues.apache.org/jira/browse/HIVE-4256.

Limitations

Empty tables might report their description without datatype information. To work around this problem when importing, you can exclude empty tables, or use the **useDatabaseMetaData** option.

9.5.10. HSQL Translator (hsql)

Also see common JDBC Translators information.

The HSQL Translator, known by the type name ***hsql***, is for use with HSQLDB 1.7 or later.

9.5.11. Informix translator (informix)

Also see common JDBC Translators information.

The Informix translator, known by the type name ***informix***, is for use with any Informix version.

Known issues

TEIID-3808

The Informix driver's handling of timezone information is inconsistent, even if the **databaseTimezone** translator property is set. Verify that the Informix server and the application server are in the same time zone.

9.5.12. Ingres translators (ingres / ingres93)

Also see common JDBC Translators information.

You can use one of the following Ingres translators, depending on your Ingres version:

ingres

The Ingres translator, known by the type name ***ingres***, is for use with Ingres 2006 or later.

ingres93

The Ingres93 translator, known by the type name ***ingres93***, is for use with Ingres 9.3 or later.

9.5.13. Intersystems Caché translator (intersystems-cache)

Also see common JDBC Translators information.

The Intersystem Caché translator, known by the type name ***intersystems-cache***, is for use with Intersystems Caché Object database (relational aspects only).

9.5.14. JDBC ANSI translator (jdbc-ansi)

Also see common JDBC Translators information.

The JDBC ANSI translator, known by the type name ***jdbc-ansi***, works with most of the SQL constructs used in Data Virtualization, except for row LIMIT/OFFSET and EXCEPT/INTERSECT. It translates source SQL into ANSI compliant syntax. This translator should be used when another more specific type is not available. If source exceptions arise due to the use of incompatible SQL constructs, then consider using the JDBC simple translator to further restrict capabilities, or create a custom translator. For more information, see the Custom Translator documentation in the Teiid community .

9.5.15. JDBC simple translator (jdbc-simple)

Also see common JDBC Translators information.

The JDBC Simple translator, known by the type name ***jdbc-simple***, is the same as the jdbc-ansi-translator, except that, to provide maximum compatibility, it does not handle most pushdown constructs.

9.5.16. Microsoft Access translators

Also see common JDBC Translators information.

access

The Microsoft Access translator known by the type name ***access*** is for use with Microsoft Access 2003 or later via the JDBC-ODBC bridge.
If you are using the default native metadata import, or the Data Virtualization connection importer, the importer defaults to **importKeys=false** and **excludeTables=.[.]MSys.** to avoid issues with the metadata provided by the JDBC ODBC bridge. You might need to adjust these values if you use a different JDBC driver.

ucanaccess

The Microsoft Access translator known by the type name ***ucanaccess*** is for use with Microsoft Access 2003 or later via the UCanAccess driver.

9.5.17. Microsoft SQL Server translator (sqlserver)

Also see common JDBC translators information.

The Microsoft SQL Server translator, known by the type name ***sqlserver***, is for use with SQL Server 2000 or later. A SQL Server JDBC driver version 2.0 or later (or a compatible driver such as, JTDS 1.2 or later) should be used. The SQL Server **DatabaseVersion** property can be set to **2000**, **2005**, **2008**, or **2012**, but otherwise expects a standard version number, for example, **10.0**.

Sequences

With Data Virtualization 8.5+, sequence operations may be modeled as source functions.

With Data Virtualization 10.0+, sequences may be imported automatically import properties.

Example: Sequence native query

```
CREATE FOREIGN FUNCTION seq_nextval () returns integer OPTIONS ("teiid_rel:native-query"
'NEXT VALUE FOR seq');
```

Execution properties

SQL Server specific execution properties:

JtdsDriver

Specifies that use of the open source JTDS driver. Defaults to false.

9.5.18. MySQL translator (mysql/mysql5)

Also see common JDBC translators information.

You can use the following translators with MySQL and MariaDB:

mysql

The MySQL translator, known by the type name ***mysql***, is for use with MySQL version 4.x.

mysql5

The MySQL5 translator, known by the type name ***mysql5***, is for use with MySQL version 5 or later. The translator also works with other compatible MySQL derivatives, such as MariaDB.

Usage

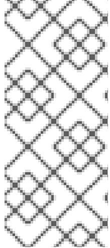
The MySQL translators expect the database or session to be using ANSI mode. If the database is not using ANSI mode, you can set ANSI mode on the pool by submitting the following initialization query:

```
set SESSION sql_mode = 'ANSI'
```

When data includes null timestamp values, Data Virtualization generates the following conversion error: **0000-00-00 00:00:00 cannot be converted to a timestamp**. To avoid error, if you expect data with null timestamp values, set the connection property **zeroDateTimeBehavior=convertToNull**.

WARNING

If you must retrieve large result sets, consider setting the connection property **useCursorFetch=true**. Otherwise MySQL will fully fetch result sets into memory on the Data Virtualization instance.



NOTE

MySQL reports TINYINT(1) columns as a JDBC BIT type - however the value range is not actually restricted and may cause issues if for example you are relying on -1 being recognized as a true value. If not using the native importer, change the BOOLEAN columns in the affected source to have a native type of "TINYINT(1)" rather than BIT so that the translator can appropriately handle the Boolean conversion.

9.5.19. Netezza translator (netezza)

Also see common JDBC translators information.

The Netezza translator, known by the type name ***netezza***, is for use with any version of the IBM Netezza appliance.

Usage

The current vendor-supplied JDBC driver for Netezza performs poorly with single transactional updates. It is best to perform batched updates whenever possible.

Execution properties

Netezza-specific execution properties:

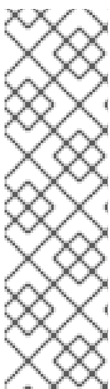
SqlExtensionsInstalled

Indicates that SQL extensions, including the ability to process Netezza **REGEXP_LIKE** functions, are installed. All other REGEXP functions are then available as pushdown functions. Defaults to **false**.

9.5.20. Oracle translator (oracle)

Also see common JDBC translators information.

The Oracle translator, known by the type name ***oracle***, is for use with Oracle Database 9i or later.



NOTE

The Oracle-provided JDBC driver uses large amounts of memory. Because the driver caches a high volume of data in the buffer, problems can occur on computers that lack sufficient memory allocation.

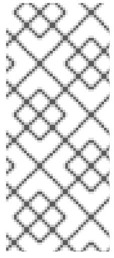
For more information, see the following resources:

- Teiid issue.
- Oracle whitepaper.

Importer properties

useGeometryType

Use the Data Virtualization Geometry type when importing columns with a source type of SDO_GEOMETRY. Defaults to false.



NOTE

Metadata import from Oracle may be slow. It is recommended that at least a schema name filter is specified. There is also the **useFetchSizeWithLongColumn=true** connection property that can increase the fetch size for metadata queries. It significantly improves the metadata load process, especially when there are a large number of tables in a schema.

Execution properties

OracleSuppliedDriver

Indicates that the Oracle supplied driver (typically prefixed by ojdbc) is being used. Defaults to true. Set to **false** when using DataDirect or other Oracle JDBC drivers.

Oracle-specific metadata

Sequences

You can use sequences with the Oracle translator. You can model a sequence as a table with a name in source of DUAL, and setting column names in the source set to **<sequence name>.[nextval|currval]**
With Data Virtualization 10.0+, you can import sequences automatically.

For more information, see *Importer properties* in JDBC translators. Data Virtualization 8.4 and Prior Oracle Sequence DDL

```
CREATE FOREIGN TABLE seq (nextval integer OPTIONS (NAMEINSOURCE 'seq.nextval'), currval
integer options (NAMEINSOURCE 'seq.currval') ) OPTIONS (NAMEINSOURCE 'DUAL')
```

With Data Virtualization 8.5 it's no longer necessary to rely on a table representation and Oracle-specific handling for sequences.

For information about representing **currval** and **nextval** as source functions, see DDL metadata for schema objects

8.5 Example: Sequence native query

```
CREATE FOREIGN FUNCTION seq_nextval () returns integer OPTIONS ("teiid_rel:native-query"
'seq.nextval');
```

You can also use a sequence as the default value for insert columns by setting the column to autoincrement, and setting the name in source to **<element name>:SEQUENCE=<sequence name>.<sequence value>**.

Rownum

A **rownum** column can also be added to any Oracle physical table to enable use of the rownum pseudo-column. A rownum column should have a name in source of **rownum**. These rownum columns do not have the same semantics as the Oracle rownum construct so care must be taken in their usage.

Out parameter result set

Out parameters for procedures may also be used to return a result set, if this is not represented correctly by the automatic import you need to manually create a result set and represent the output parameter with native type **REF CURSOR**.

DDL for out parameter result set

```
create foreign procedure proc (in x integer, out y object options (native_type 'REF CURSOR'))
returns table (a integer, b string)
```

Geospatial functions

You can use the following geospatial functions with the translator for Oracle:

Relate = sdo_relate

```
CREATE FOREIGN FUNCTION sdo_relate (arg1 string,  arg2 string,  arg3 string) RETURNS string;
CREATE FOREIGN FUNCTION sdo_relate (arg1 Object,  arg2 Object,  arg3 string) RETURNS
string;
CREATE FOREIGN FUNCTION sdo_relate (arg1 string,  arg2 Object,  arg3 string) RETURNS string;
CREATE FOREIGN FUNCTION sdo_relate (arg1 Object,  arg2 string,  arg3 string) RETURNS string;
```

Nearest_Neighbor = sdo_nn

```
CREATE FOREIGN FUNCTION sdo_nn (arg1 string,  arg2 Object,  arg3 string,  arg4 integer)
RETURNS string;
CREATE FOREIGN FUNCTION sdo_nn (arg1 Object,  arg2 Object,  arg3 string,  arg4 integer)
RETURNS string;
CREATE FOREIGN FUNCTION sdo_nn (arg1 Object,  arg2 string,  arg3 string,  arg4 integer)
RETURNS string;
```

Within_Distance = sdo_within_distance

```
CREATE FOREIGN FUNCTION sdo_within_distance (arg1 Object,  arg2 Object,  arg3 string)
RETURNS string;
CREATE FOREIGN FUNCTION sdo_within_distance (arg1 string,  arg2 Object,  arg3 string)
RETURNS string;
CREATE FOREIGN FUNCTION sdo_within_distance (arg1 Object,  arg2 string,  arg3 string)
RETURNS string;
```

Nearest_Neigher_Distance = sdo_nn_distance

```
CREATE FOREIGN FUNCTION sdo_nn_distance (arg integer) RETURNS integer;
```

Filter = sdo_filter

```
CREATE FOREIGN FUNCTION sdo_filter (arg1 Object,  arg2 string,  arg3 string) RETURNS string;
CREATE FOREIGN FUNCTION sdo_filter (arg1 Object,  arg2 Object,  arg3 string) RETURNS string;
CREATE FOREIGN FUNCTION sdo_filter (arg1 string,  arg2 object,  arg3 string) RETURNS string;
```

Pushdown functions

Depending on the Oracle version, the Oracle translator, registers the following non-geospatial pushdown functions with the engine:

TRUNC

Both numeric and timestamp versions.

LISTAGG

Requires the Data Virtualization SQL syntax "LISTAGG(arg [, delim] ORDER BY ...)"

SQLXML

If you need to retrieve SQLXML values from Oracle and are getting oracle.xdb.XMLType or OPAQUE instances instead, you make the following changes:

- Use client driver version 11, or later.
- Place the **xdb.jar** and **xmlparserv2.jar** files in the classpath.
- Set the system property **oracle.jdbc.getObjectReturnsXMLType="false"**.
  For more information, see the Oracle documentation.

9.5.21. PostgreSQL translator (postgresql)

Also see common JDBC translators information.

The PostgreSQL translator, known by the type name ***postgresql***, is for use with the following PostgreSQL client and server versions: * Client – 8.0 or later * Server – 7.1 or later.

Execution properties

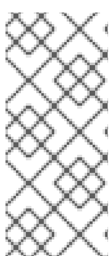
PostgreSQL-specific execution properties:

PostGisVersion

Indicates the PostGIS version in use. Defaults to 0, which means that PostGIS is not installed. Will be set automatically if the database version is not set.

ProjSupported

Boolean that indicates if the PostGis version supports PROJ coordinate transformation software. Will be set automatically if the database version is not set.



NOTE

Some driver versions of PostgreSQL will not associate columns to "INDEX" type tables. The current version of Data Virtualization omits such tables automatically.

Older versions of Data Virtualization may need the importer.tableType property or other filtering set.

9.5.22. PrestoDB translator (prestodb)

Also see common JDBC translators information.

The PrestoDB translator, known by the type name ***prestodb***, exposes querying functionality to Presto data sources. In data integration respect, PrestoDB has capabilities that are similar to Data Virtualization, however it goes beyond in terms of distributed query execution with multiple worker nodes. Data Virtualization's execution model is limited to single execution node and focuses more on pushing the query down to sources. Data Virtualization provides more complete querying capabilities and many enterprise features.

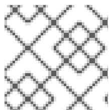
Capabilities

You can use the PrestoDB translator only with **SELECT** statements. The translator provides a restricted set of capabilities.

Because PrestoDB exposes a relational model, Data Virtualization can use it as it does other RDBMS sources, such as Oracle, Db2, and so forth. For information about configuring PrestoDB, see the Presto documentation.

TIP

In SQL JOIN operations, PrestoDB does not support multiple **ORDER BY** columns well. If errors occur during JOIN operations that involve more than one **ORDER BY** column, set the translator property **supportsOrderBy** to disable the use of the **ORDER BY** clause.

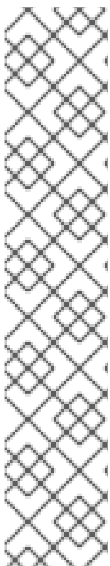


NOTE

Some versions of Presto generate errors when you include null values in subqueries.

TIP

PrestoDB does not support transactions. To overcome issues caused by this limitation, define the data source as non-transactional.



NOTE

By default, every catalog in PrestoDB has an **information_schema**. If you have to configure multiple catalogs, duplicate table errors can cause deployment of a virtual database to fail. To prevent duplicate table errors, use import options to filter the schemas.

If you want to configure multiple Presto catalogs, set one of the following import options to filter the schemas and tables in the source:

- Set **catalog** to a specific catalog name to match the name of the source catalog in Presto.
- Set **schemaName** to a regular expression to filter schemas by matching result.
- Set **excludeTables** to a regular expression to filter tables by matching results.

9.5.23. Redshift translator (redshift)

Also see common JDBC translators information.

The Redshift translator, known by the type name ***redshift***, is for use with the Amazon Redshift database. This translator is an extension of the PostgreSQL translator and inherits its options.

9.5.24. SAP HANA translator (hana)

Also see common JDBC translators information.

The SAP HANA translator, known by the name of ***hana***, is for use with SAP HANA.

Known issues

TEIID-3805

The pushdown of the SUBSTRING function is inconsistent with the Data Virtualization SUBSTRING function when the FROM index exceeds the length of the string. SAP HANA will return an empty string, while Data Virtualization produces a null value.

9.5.25. SAP IQ translator (sap-iq)

Also see common JDBC translators information.

The SAP IQ translator, known by the type name ***sap-iq***, is for use with SAP IQ version 15.1 or later. The translator name ***sybaseiq*** has been deprecated.

9.5.26. Sybase translator (sybase)

Also see common JDBC Translators information.

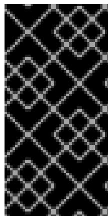
The Sybase translator, known by the type name ***sybase***, is for use with SAP ASE (Adaptive Server Enterprise), formerly known as Sybase SQL Server, version 12.5 or later.

If you use the default native import, you can avoid exceptions during the retrieval of system table information, if you specify import properties. If errors occur when retrieving table information, specify a **schemaName** or **schemaPattern**, or use **excludeTables** to exclude system tables. For more information about using import properties, see *Importer properties* in JDBC translators.

If the name in the source metadata contains quoted identifiers (such as reserved words, or words that contain characters that would not otherwise be allowed), and you are using a jConnect Sybase driver, you must first configure the connection pool to enable **quoted_identifier**:

Example: Driver URL with SQLINITSTRING

```
jdbc:sybase:Tds:host.at.some.domain:5000/db_name?SQLINITSTRING=set quoted_identifier on
```



IMPORTANT

If you are using a jConnect Sybase driver and will target the source for dependent joins, set the **JCONNECT_VERSION** to **6** or later to increase the number of values that the translator can send. If you do not set the **JCONNECT_VERSION**, an exception occurs with statements that have more than 481 bind values.

Example: Driver URL with JCONNECT_VERSION

```
jdbc:sybase:Tds:host.at.some.domain:5000/db_name?SQLINITSTRING=set quoted_identifier
on&JCONNECT_VERSION=6
```

Execution properties specific to Sybase

JtdsDriver_

Indicates that the open source JTDS driver is being used. Defaults to false.

9.5.27. Data Virtualization translator (teiid)

Also see common JDBC translators information.

Use the Teiid translator, known by the type name ***teiid***, when creating a virtual database from a Teiid data source.

9.5.28. Teradata translator (teradata)

Also see common JDBC translators information.

The Teradata translator, known by the type name ***teradata***, is for use with Teradata Database V2R5.1 or later.

By default, Teradata driver version 15, adjusts date, time, and timestamp values to match the Data Virtualization server timezone. To remove this adjustment, set the translator **DatabaseTimezone** property to GMT or whatever the Teradata server defaults to.

9.5.29. Vertica translator (vertica)

Also see common JDBC translators information.

The Vertica translator, known by the type name ***vertica***, is for use with Vertica 6 or later.

9.6. LOOPBACK TRANSLATOR

The Loopback translator, known by the type name *loopback*, provides a quick testing solution. It works with all SQL constructs and returns default results, with some configurable behavior.

Table 9.5. Execution properties

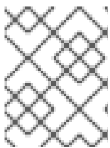
Name	Description	Default
ThrowError	**true** to always throw an error.	false
RowCount	Rows returned for non-update queries.	1
WaitTime	Wait randomly up to this number of milliseconds with each source query.	0

Name	Description	Default
PollIntervalInMilli	If positive, results will be **asynchronously** returned – that is a **DataNotAvailableException** will be thrown initially and the engine will wait the poll interval before polling for the results.	-1
DelegateName	Set to the name of the translator which is to be mimicked.	na

You can also use the Loopback translator to mimic how a real source query would be formed for a given translator (although loopback will still return dummy data that might not be useful for your situation). To enable this behavior, set the **DelegateName** property to the name of the translator that you want to mimic. For example, to disable all capabilities, set the **DelegateName** property to **jdbc-simple**.

9.7. MICROSOFT EXCEL TRANSLATOR

The Microsoft Excel Translator, known by the type name *excel*, exposes querying functionality to a Microsoft Excel document. This translator provides an easy way read a Excel spreadsheet and provide the contents of the spreadsheet in a tabular form that can be integrated with other sources in Data Virtualization.



NOTE

This translator works on all platforms, including Windows and Linux. The translator uses Apache POI libraries to access the Excel documents which are platform independent.

Translation mapping

The following table describes how Excel translator interprets the data in Excel document into relational terms.

Excel Term	Relational term
Workbook	schema
Sheet	Table
Row	Row of data
Cell	Column Definition or Data of a column

The Excel translator provides a "source metadata" feature, where for a given Excel workbook, it can introspect and build the schema based on the worksheets that are defined within it. There are options available to detect header columns and data columns in a worksheet to define the correct metadata of a table.

DDL example

The following example shows how to expose an Excel spreadsheet in a virtual database.

```
CREATE DATABASE excelvdb;
USE DATABASE excelvdb;
CREATE SERVER connector FOREIGN DATA WRAPPER excel OPTIONS ("resource-name"
'java:/fileDS');
CREATE SCHEMA excel SERVER connector;
SET SCHEMA excel;
IMPORT FROM SERVER connector INTO excel OPTIONS (
   "importer.headerRowNumber" '1',
   "importer.ExcelFileName" 'names.xls');
```

Headers in document

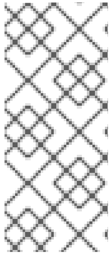
If the Excel document contains headers, you can guide the import process to select the cell headers as the column names in the table creation process. For information about defining import properties, see the following table, and also see *Importer Properties* in JDBC translators.

Import properties

Import properties guide the schema generation part during the deployment of the VDB. This can be used in a native import.

Property Name	Description	Default
importer.excelFileName	Defines the name of the Excel Document to import metadata. This can be defined as a file pattern (*.xls), however when defined as pattern all files must be of same format, and the translator will choose an arbitrary file to import metadata from. Use file patterns to read data from multiple Excel documents in the same directory. In the case of a single file, specify the absolute name.	Required
importer.headerRowNumber	Defines the cell header information to be used as column names.	Optional. Default is first data row of sheet
importer.dataRowNumber	Defines the row number where the data rows start.	Optional. Default is first data row of sheet.

To enable information in the Excel spreadsheet to be interpreted correctly, it is best to define all the preceding importer properties.



NOTE

Purely numerical cells in a column contain containing mixed types will have a string form matching their decimal representation, thus integral values will have **.0** appended. If you need the exact text representation, then the cell must be a string value. You can force a string value by preceding the numeric text of a cell with a single quote ('), or a single space.

Translator extension properties

- Excel specific execution properties

  FormatStrings

  Format non-string cell values in a string column according to the worksheet format. Defaults to false.

- Metadata extension properties
  Properties that are defined on schema artifacts, such as Table, Column, Procedure and so forth. These properties describe how the translator interacts with or interprets source systems. All the properties are defined with the following namespace: **"http://www.teiid.org/translator/excel/2014[http://www.teiid.org/translator/excel/2014\]"**, which also has a recognized alias **teiid_excel**.

Property Name	Schema item property belongs to	Description	Mandatory
FILE	Table	Defines Excel Document name or name pattern (*.xls). File pattern can be used to read data from multiple files.	Yes
FIRST_DATA_ROW_NUMBER	Table	Defines the row number where records start in the sheet (applies to every sheet).	Optional
CELL_NUMBER	Column of Table	Defines cell number to use for reading data of particular column.	Yes

The following example shows a table that is defined by using the extension metadata properties.

```
CREATE DATABASE excelvdb;
USE DATABASE excelvdb;
CREATE SERVER connector FOREIGN DATA WRAPPER excel OPTIONS ("resource-name"
'java:/fileDS');
CREATE SCHEMA excel SERVER connector;
SET SCHEMA excel;
CREATE FOREIGN TABLE Person (
	ROW_ID integer OPTIONS (SEARCHABLE 'All_Except_Like',
"teiid_excel:CELL_NUMBER" 'ROW_ID'),
	FirstName string OPTIONS (SEARCHABLE 'Unsearchable', "teiid_excel:CELL_NUMBER"
```

```
'1'),
          LastName string OPTIONS (SEARCHABLE 'Unsearchable', "teiid_excel:CELL_NUMBER"
'2'),
          Age integer OPTIONS (SEARCHABLE 'Unsearchable', "teiid_excel:CELL_NUMBER" '3'),
          CONSTRAINT PK0 PRIMARY KEY(ROW_ID)
        ) OPTIONS ("NAMEINSOURCE" 'Sheet1',"teiid_excel:FILE" 'names.xlsx',
"teiid_excel:FIRST_DATA_ROW_NUMBER" '2')
```

Extended capabilities using ROW_ID column

If you define a column that has extension metadata property **CELL_NUMBER** with value **ROW_ID**, then that column value contains the row information from Excel document. You can mark this column as Primary Key. You can use this column in **SELECT** statements with a restrictive set of capabilities including: comparison predicates, **IN** predicates and **LIMIT**. All other columns cannot be used as predicates in a query.

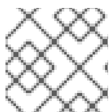
TIP

Importing source metadata is not the only way to create the schema of an Excel document. You can also create a source table manually, and then add the extension properties that you need to create a fully functional model. Metadata imports result in schema models similar to the one in the preceding example.

The Excel translator processes updates with the following limitations:

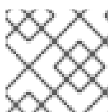
- The **ROW_ID** cannot be directly modified or used as an insert value.
- UPDATE and INSERT values must be literals.
- UPDATEs are not transactional. That is, the write lock is held while the file is written, but not throughout the entire update. As a result, it is possible for one update to overwrite another.

The **ROW_ID** of an inserted row can be returned as a generated key.



NATIVE QUERIES

This feature is not applicable for the Excel translator.



DIRECT QUERY PROCEDURE

This feature is not applicable for the Excel translator.

9.8. MONGODB TRANSLATOR

The MongoDB translator, known by the type name *mongodb*, provides a relational view of data that resides in a MongoDB database. This translator is capable of converting Data Virtualization SQL queries into MongoDB based queries. It provides for a full range of SELECT, INSERT, UPDATE and DELETE calls.

MongoDB is a document based "schema-less" database with it own query language. It does not map perfectly with relational concepts or the SQL query language. More and more systems are using NOSQL stores such as MongoDB to improve scalability and performance. For example, applications like storing audit logs, or managing web site data, are well-suited to MongoDB, and do not require the structure of relational databases. MongoDB uses JSON documents as its primary storage unit, and those documents

can have additional embedded documents inside the parent document. By using embedded documents, MongoDB co-locates related information to achieve de-normalization that typically requires either duplicate data or joins to achieve querying in a relational database.

For MongoDB to work with Data Virtualization the challenge for the MongoDB translator is to design a MongoDB store that can achieve the balance between relational and document based storage. The advantages of "schema-less" design are great at development time. But "schema-less" design can pose problems during migration between application versions, and when querying data, and making effective use of the returned information.

Since it is hard and may be impossible in certain situations to derive a schema based on existing the MongoDB collection(s), Data Virtualization approaches the problem in reverse compared to other translators. When working with MongoDB, Data Virtualization requires you to define the MongoDB schema upfront, by using Data Virtualization metadata. Because Data Virtualization only allows relational schema as its metadata, you must define your MongoDB schema in relational terms, using tables, procedures, and functions. For the purposes of MongoDB, the Data Virtualization metadata has been extended to provide extension properties that can be defined on a table to convert it into a MongoDB based document. These extension properties let you define how a MongoDB document is structured and stored. Based on the relationships (primary-key, foreign-key) that are defined on a table, and their cardinality (ONE-to-ONE, ONE-to-MANY, MANY-to-ONE), relations between tables are mapped such that related information can be embedded along with the parent document for co-location (as mentioned earlier in this topic). Thus, a relational schema-based design, but document-based storage in MongoDB.

Who is the primary audience for the MongoDB translator?

The above may not satisfy every user's needs. The document structure in MongoDB can be more complex than what Data Virtualization can currently define. We hope this will eventually catch up in future versions of Data Virtualization. This is currently designed for:

- Users who are using relational databases and would like to move/migrate their data to MongoDB to take advantage of scaling and performance without modifying end user applications that they currently run.
- Users who are seasoned SQL developers, but do not have experience with MongoDB. This provides a low barrier of entry compared to using MongoDB directly as an application developer.
- Users who want to integrate MongoDB-based data with data from other enterprise data sources.

Usage

The name of the translator to use in a virtual database DDL is "*mongodb*". For example:

```
CREATE DATABASE nothwind;
USE DATABASE nothwind;
CREATE SERVER local FOREIGN DATA WRAPPER mongodb OPTIONS ("resource-name"
'java:/mongoDS');
CREATE SCHEMA northwind SERVER local;

SET SCHEMA northwind;
IMPORT FROM SERVER local INTO northwind;
```

The MongoDB translator can derive the metadata based on existing document collections in some scenarios. However, when working with complex documents the interpretation of metadata can be inaccurate. In such cases, you must define the metadata. For example, you can define a schema using DDL, as shown in the following example:

```
<vdb name="nothwind" version="1">
  <model name="northwind">
    <source name="local" translator-name="mongodb" connection-jndi-name="java:/mongoDS"/>
      <metadata type="DDL"><![CDATA[
        CREATE FOREIGN TABLE  Customer (
          customer_id integer,
          FirstName varchar(25),
          LastName varchar(25)
        ) OPTIONS(UPDATABLE 'TRUE');
      ]]> </metadata>
  </model>
<vdb>
```

When the following INSERT operation is executed against a table using Data Virtualization, the MongoDB translator creates a document in the MongoDB:

```
INSERT INTO Customer(customer_id, FirstName, LastName) VALUES (1, 'John', 'Doe');
```

```
{
 _id: ObjectID("509a8fb2f3f4948bd2f983a0"),
 customer_id: 1,
 FirstName: "John",
 LastName: "Doe"
}
```

If a PRIMARY KEY is defined on the table, then that column name is automatically used as **"_id"** field in the MongoDB collection, and then the document structure is stored in the MongoDB, as shown in the following examples:

```
CREATE FOREIGN TABLE  Customer (
  customer_id integer PRIMARY KEY,
  FirstName varchar(25),
  LastName varchar(25)
) OPTIONS(UPDATABLE 'TRUE');
```

```
{
 _id: 1,
 FirstName: "John",
 LastName: "Doe"
}
```

If you defined the composite PRIMARY KEY on Customer table, the document structure that results is shown in the following example:

```
CREATE FOREIGN TABLE  Customer (
  customer_id integer,
  FirstName varchar(25),
  LastName varchar(25),
  PRIMARY KEY (FirstName, LastName)
) OPTIONS(UPDATABLE 'TRUE');
```

```
{
 _id: {
```

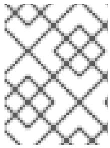
```
        FirstName: "John",
        LastName:  "Doe"
      },
   customer_id: 1,
}
```

Data types

The MongoDB translator provides automatic mapping of Data Virtualization data types into MongoDB data types, including BLOBS, CLOBS and XML. The LOB mapping is based on GridFS in MongoDB. Arrays are in the following form:

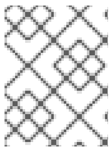
```
{
  _id: 1,
  FirstName: "John",
  LastName: "Doe"
  Score: [89, "ninety", 91.0]
}
```

Users can get individual items in the array using the function **array_get**, or can transform the array into tabular structure using ARRAYTABLE.



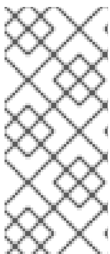
NOTE

Note that even though embedded documents can also be in arrays, the handling of embedded documents is different from array with scalar values.



NOTE

The translator does not work with regular Expressions, MongoDB::Code, MongoDB::MinKey, MongoDB::MaxKey, and MongoDB::OID.



NOTE

In documents that contain values of mixed types for the same key, you must mark the column as unsearchable, or MongoDB will not correctly match predicates against the column. A key is used as a mixed type of it is represented as a string value in one document, and an integer in another. For more information, see the **importer.sampleSize property** in the following table.

Importer Properties

Importer properties define the behavior of the translator during the metadata import from the physical source.

Importer Properties

Name	Description	Default
excludeTables	Regular expression to exclude the tables from import.	null

Name	Description	Default
includeTables	Regular expression to include the tables from import.	null
sampleSize	Number of documents to sample to determine the structure. If documents have different fields, or fields with different types, this should be greater than 1.	1
fullEmbeddedNames	Whether to prefix embedded table names with their parents, e.g. parent_embedded. If false the name of the table will just be the name of the field - which may lead to conflicts with existing tables or other embedded tables.	false

MongoDB metadata extension properties for building complex documents

Using the preceding DDL, or any other metadata facility, you can map a table in a relational store into a document in MongoDB. However, to make effective use of MongoDB, you must be able to build complex documents that can co-locate related information, so that data can queried in a single MongoDB query. Unlike a relational database, you cannot run join operations in MongoDB. As as a result, unless you can build complex documents, you would have to issue multiple queries to retrieve data and then join it manually. The power of MongoDB comes from its "embedded" documents, its support for complex data types, such as arrays, and its use of an aggregation framework to query them. This translator provides a way to achieve the goals.

When you do not define the complex embedded documents in MongoDB, Data Virtualization can step in for join processing and provide that functionality. However, if you want to make use of the power of MongoDB itself in querying the data and avoid bringing the unnecessary data and improve performance, you need to look into building these complex documents.

MongoDB translator defines two additional metadata properties along with other Teiid metadata properties to aid in building the complex "embedded" documents. For more information about Data Virtualization schema metadata, see Section 2.2, “DDL metadata for schema objects” . You can use the following metadata properties in your DDL:

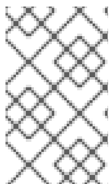
teiid_mongo:EMBEDDABLE

Means that data defined in this table is allowed to be included as an "embeddable" document in **any** parent document. The parent document is referenced by the foreign key relationships. In this scenario, Data Virtualization maintains more than one copy of the data in MongoDB store, one in its own collection, and also a copy in each of the parent tables that have relationship to this table. You can even nest embeddable table inside another embeddable table with some limitations. Use this property on table, where table can exist, encompass all its relations on its own. For example, a "Category" table that defines a "Product"'s category is independent of Product, which can be embeddable in "Products" table.

teiid_mongo:MERGE

Means that data of this table is merged with the defined parent table. There is only a single copy of the data that is embedded in the parent document. Parent document is defined using the foreign key relationships.

Using the above properties and FOREIGN KEY relationships, we will illustrate how to build complex documents in MongoDB.



USAGE

A given table can contain either the **teiid_mongo:EMBEDDABLE** property or the **teiid_mongo:MERGE** property defining the type of nesting in MongoDB. You cannot use both properties within one table.

ONE-2-ONE Mapping

If your current DDL structure representing ONE-2-ONE relationship is like

```
CREATE FOREIGN TABLE  Customer (
    CustomerId integer PRIMARY KEY,
    FirstName varchar(25),
    LastName varchar(25)
) OPTIONS(UPDATABLE 'TRUE');

CREATE FOREIGN TABLE Address (
    CustomerId integer,
    Street varchar(50),
    City varchar(25),
    State varchar(25),
    Zipcode varchar(6),
    FOREIGN KEY (CustomerId) REFERENCES Customer (CustomerId)
 ) OPTIONS(UPDATABLE 'TRUE');
```

By default, this will produce two different collections in MongoDB, like with sample data it will look like

```
Customer
{
  _id: 1,
  FirstName: "John",
  LastName: "Doe"
}

Address
{
  _id: ObjectID("..."),
   CustomerId: 1,
   Street: "123 Lane"
   City: "New York",
   State: "NY"
   Zipcode: "12345"
}
```

You can enhance the storage in MongoDB to a single collection by using **teiid_mongo:MERGE** extension property on the table's OPTIONS clause.

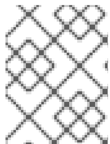
```
CREATE FOREIGN TABLE  Customer (
    CustomerId integer PRIMARY KEY,
    FirstName varchar(25),
    LastName varchar(25)
) OPTIONS(UPDATABLE 'TRUE');
```

```
CREATE FOREIGN TABLE Address (
    CustomerId integer PRIMARY KEY,
    Street varchar(50),
    City varchar(25),
    State varchar(25),
    Zipcode varchar(6),
    FOREIGN KEY (CustomerId) REFERENCES Customer (CustomerId)
 ) OPTIONS(UPDATABLE 'TRUE', "teiid_mongo:MERGE" 'Customer');
```

this will produce single collection in MongoDB, like

```
Customer
{
  _id: 1,
  FirstName: "John",
  LastName: "Doe",
  Address:
    {
      Street: "123 Lane",
      City: "New York",
      State: "NY",
      Zipcode: "12345"
    }
}
```

With the above both tables are merged into a single collection that can be queried together using the JOIN clause in the SQL command. Since the existence of child/additional record has no meaning with out parent table using the "*teiid_mongo:MERGE*" extension property is right choice in this situation.



NOTE

The Foreign Key defined on a child table must refer to Primary Keys on both the parent and child tables to form a One-2-One relationship.

ONE-2-MANY Mapping.

Typically there can be more than two (2) tables involved in this relationship. If MANY side is only associated **single** table, then use **teiid_mongo:MERGE** property on MANY side of table and define ONE as the parent. If associated with more than single table then use **teiid_mongo:EMBEDDABLE**.

For example, if you define a virtual database as in the following DDL:

```
CREATE FOREIGN TABLE  Customer (
    CustomerId integer PRIMARY KEY,
    FirstName varchar(25),
    LastName varchar(25)
) OPTIONS(UPDATABLE 'TRUE');

CREATE FOREIGN TABLE  Order (
    OrderID integer PRIMARY KEY,
    CustomerId integer,
    OrderDate date,
```

```
    Status integer,
    FOREIGN KEY (CustomerId) REFERENCES Customer (CustomerId)
) OPTIONS(UPDATABLE 'TRUE');
```

then a Single Customer can have MANY Orders. There are two options to define the how we store the MongoDB document. If in your schema, the Customer table's CustomerId is **only** referenced in Order table (i.e. Customer information used for only Order purposes), you can use

```
CREATE FOREIGN TABLE  Customer (
    CustomerId integer PRIMARY KEY,
    FirstName varchar(25),
    LastName varchar(25)
) OPTIONS(UPDATABLE 'TRUE');

CREATE FOREIGN TABLE  Order (
    OrderID integer PRIMARY KEY,
    CustomerId integer,
    OrderDate date,
    Status integer,
    FOREIGN KEY (CustomerId) REFERENCES Customer (CustomerId)
) OPTIONS(UPDATABLE 'TRUE', "teiid_mongo:MERGE" 'Customer');
```

that will produce a single document for Customer table like

```
{
  _id: 1,
  FirstName: "John",
  LastName: "Doe",
  Order:
  [
    {
     _id: 100,
      OrderDate: ISODate("2000-01-01T06:00:00Z")
      Status: 2
    },
    {
     _id: 101,
      OrderDate: ISODate("2001-03-06T06:00:00Z")
      Status: 5
    }
    ...
   ]
}
```

If Customer table is referenced in more tables other than Order table, then use "teiid_mongo:EMBEDDABLE" property

```
CREATE FOREIGN TABLE Customer (
    CustomerId integer PRIMARY KEY,
    FirstName varchar(25),
    LastName varchar(25)
) OPTIONS(UPDATABLE 'TRUE', "teiid_mongo:EMBEDDABLE" 'TRUE');

CREATE FOREIGN TABLE Order (
    OrderID integer PRIMARY KEY,
```

```
        CustomerId integer,
        OrderDate date,
        Status integer,
        FOREIGN KEY (CustomerId) REFERENCES Customer (CustomerId)
    ) OPTIONS(UPDATABLE 'TRUE');

    CREATE FOREIGN TABLE Comments (
        CommentID integer PRIMARY KEY,
        CustomerId integer,
        Comment varchar(140),
        FOREIGN KEY (CustomerId) REFERENCES Customer (CustomerId)
    ) OPTIONS(UPDATABLE 'TRUE');
```

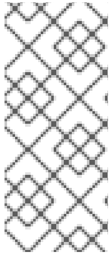
This creates three different collections in MongoDB.

```
Customer
{
  _id: 1,
  FirstName: "John",
  LastName: "Doe"
}

Order
{
  _id: 100,
  CustomerId: 1,
  OrderDate: ISODate("2000-01-01T06:00:00Z")
  Status: 2
  Customer:
   {
     FirstName: "John",
     LastName: "Doe"
   }
}

Comment
{
  _id: 12,
  CustomerId: 1,
  Comment: "This works!!!"
  Customer:
   {
     FirstName: "John",
     LastName: "Doe"
   }
}
```

Here as you can see the Customer table contents are embedded along with other table’s data where they were referenced. This creates duplicated data where multiple of these embedded documents are managed automatically in the MongoDB translator.

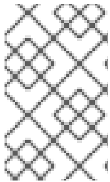


NOTE

All the SELECT, INSERT, DELETE operations that are generated against the tables with "teiid_mongo:EMBEDDABLE" property are atomic, except for UPDATES, as there can be multiple operations involved to update all the copies. Since there are no transactions in MongoDB, Data Virtualization plans to provide automatic compensating transaction framework around this in future releases TEIID-2957.

MANY-2-ONE Mapping.

This is same as ONE-2-MANY, see above to define relationships.



NOTE

A parent table can have multiple "embedded" and as well as "merge" documents inside it, it not limited so either one or other. However, please note that MongoDB imposes document size is limited can not exceed 16MB.

MANY-2-MANY Mapping.

This can also mapped with combination of "teiid_mongo:MERGE" and "teiid_mongo:EMBEDDABLE" properties (partially). For example if DDL looks like

```
CREATE FOREIGN TABLE Order (
    OrderID integer PRIMARY KEY,
    OrderDate date,
    Status integer
) OPTIONS(UPDATABLE 'TRUE');

CREATE FOREIGN TABLE OrderDetail (
    OrderID integer,
    ProductID integer,
    PRIMARY KEY (OrderID,ProductID),
    FOREIGN KEY (OrderID) REFERENCES Order (OrderID),
    FOREIGN KEY (ProductID) REFERENCES Product (ProductID)
) OPTIONS(UPDATABLE 'TRUE');

CREATE FOREIGN TABLE Products (
   ProductID integer PRIMARY KEY,
   ProductName varchar(40)
) OPTIONS(UPDATABLE 'TRUE');
```

you modify the DDL like below, to have

```
CREATE FOREIGN TABLE Order (
    OrderID integer PRIMARY KEY,
    OrderDate date,
    Status integer
) OPTIONS(UPDATABLE 'TRUE');

CREATE FOREIGN TABLE OrderDetail (
    OrderID integer,
    ProductID integer,
    PRIMARY KEY (OrderID,ProductID),
    FOREIGN KEY (OrderID) REFERENCES Order (OrderID),
```

```
    FOREIGN KEY (ProductID) REFERENCES Product (ProductID)
) OPTIONS(UPDATABLE 'TRUE', "teiid_mongo:MERGE" 'Order');

CREATE FOREIGN TABLE Products (
   ProductID integer PRIMARY KEY,
   ProductName varchar(40)
) OPTIONS(UPDATABLE 'TRUE',  "teiid_mongo:EMBEDDABLE" 'TRUE');
```

That will produce a document like

```
{
  _id : 10248,
  OrderDate : ISODate("1996-07-04T05:00:00Z"),
  Status : 5
  OrderDetails : [
   {
     _id : {
             OrderID : 10248,
             ProductID : 11
             Products : {
                ProductID: 11
                ProductName: "Hammer"
             }
     }
   },
   {
     _id : {
       OrderID : 10248,
       ProductID : 14
       Products : {
           ProductID: 14
           ProductName: "Screw Driver"
       }
     }
   }
  ]
}

Products
{
   {
     ProductID: 11
     ProductName: "Hammer"
   }
   {
     ProductID: 14
     ProductName: "Screw Driver"
   }
}
```

Limitations

- Nested embedding of documents is limited due to capabilities of handling nested arrays is limited in the MongoDB. Nesting of "EMBEDDABLE" property with multiple levels is OK, however more than two levels with MERGE is not recommended. Also, you need to be caution

about not exceeding the document size of 16 MB for single row, so deep nesting is not recommended.

- JOINS between related tables, MUST use either the "EMBEDDABLE" or "MERGE" properties, otherwise the query will result in error. In order for Data Virtualization to correctly plan and work with JOINS, in the case that any two tables are **NOT** embedded in each other, use *allow-joins=false* property on the Foreign Key that represents the relation. For example:

```
CREATE FOREIGN TABLE  Customer (
    CustomerId integer PRIMARY KEY,
    FirstName varchar(25),
    LastName varchar(25)
) OPTIONS(UPDATABLE 'TRUE');

CREATE FOREIGN TABLE  Order (
    OrderID integer PRIMARY KEY,
    CustomerId integer,
    OrderDate date,
    Status integer,
    FOREIGN KEY (CustomerId) REFERENCES Customer (CustomerId) OPTIONS (allow-join
'FALSE')
) OPTIONS(UPDATABLE 'TRUE');
```

with the example above, Data Virtualization will create two collections, however when user issues query such as

```
SELECT OrderID, LastName FROM Order JOIN Customer ON Order.CustomerId =
Customer.CustomerId;
```

instead of resulting in error, the JOIN processing will happen in the Data Virtualization engine, without the above property it will result in an error.

When you use above properties and carefully design the MongoDB document structure, Data Virtualization translator can intelligently collate data based on their co-location and take advantage of it while querying.

Geo Spatial functions

MongoDB translator enables you to use geo spatial query operators in the "WHERE" clause, when the data is stored in the GeoJSon format in the MongoDB Document. The following functions are available:

```
CREATE FOREIGN FUNCTION geoIntersects (columnRef string,  type string, coordinates double[][])
RETURNS boolean;
CREATE FOREIGN FUNCTION geoWithin (ccolumnRef string,  type string, coordinates double[][])
RETURNS boolean;
CREATE FOREIGN FUNCTION near (ccolumnRef string,  coordinates double[], maxdistance
integer) RETURNS boolean;
CREATE FOREIGN FUNCTION nearSphere (ccolumnRef string, coordinates double[], maxdistance
integer) RETURNS boolean;
CREATE FOREIGN FUNCTION geoPolygonIntersects (ref string, north double, east double, west
double, south double) RETURNS boolean;
CREATE FOREIGN FUNCTION geoPolygonWithin (ref string, north double, east double, west
double, south double) RETURNS boolean;
```

a sample query looks like

```
SELECT loc FROM maps where mongo.geoWithin(loc, 'LineString', ((cast(1.0 as double), cast(2.0 as
double)), (cast(1.0 as double), cast(2.0 as double))))
```

Same functions using built-in Geometry type (the versions of the functions in the preceding list will be deprecated and removed in future versions)

```
CREATE FOREIGN FUNCTION geoIntersects (columnRef string,  geo geometry) RETURNS
boolean;
CREATE FOREIGN FUNCTION geoWithin (ccolumnRef string,  geo geometry) RETURNS boolean;
CREATE FOREIGN FUNCTION near (ccolumnRef string, geo geometry, maxdistance integer)
RETURNS boolean;
CREATE FOREIGN FUNCTION nearSphere (ccolumnRef string, geo geometry, maxdistance
integer) RETURNS boolean;
CREATE FOREIGN FUNCTION geoPolygonIntersects (ref string, geo geometry) RETURNS
boolean;
CREATE FOREIGN FUNCTION geoPolygonWithin (ref string, geo geometry) RETURNS boolean;
```

a sample query looks like

```
SELECT loc FROM maps where mongo.geoWithin(loc, ST_GeomFromGeoJSON('{"coordinates":
[[1,2],[3,4]],"type":"Polygon"}'))
```

There are various "st_geom.." methods are available in the Geo Spatial function library in Data Virtualization.

Capabilities

MongoDB translator is designed on top of the MongoDB aggregation framework. You must use a MongoDB version that the aggregation framework. Apart from SELECT queries, the MongoDB translator also works with INSERT, UPDATE and DELETE queries.

You can use the MongoDB translator with the following functions:

- Grouping.
- Matching.
- Sorting.
- Filtering.
- Limits.
- Working with LOBs stored in GridFS.
- Composite primary and foreign keys.

Native queries

MongoDB source procedures may be created using the **teiid_rel:native-query** extension. For more information, see *Parameterizable native queries* in Translators. The procedure will invoke the native-query similar to a direct procedure call with the benefits that the query is predetermined and that result column types are known, rather than requiring the use of ARRAYTABLE or similar functionality.

Direct query procedure

This feature is turned off by default because of the security risk this exposes to execute any command

against the source. To enable direct query procedures, set the execution property called **SupportsDirectQueryProcedure** to **true**. For more information, see *Override the execution properties* in Chapter 9, *Translators*.

By default the name of the procedure that executes the queries directly is called **native**. For information about how to change the default name, see *Override the execution properties* in Chapter 9, *Translators*.

The MongoDB translator provides a procedure to execute any ad-hoc aggregate query directly against the source without Data Virtualization parsing or resolving. Since the metadata of this procedure's results are not known to Data Virtualization, they are returned as an object array containing single blob at array location one(1). This blob contains the JSON document. XMLTABLE can be used construct tabular output for consumption by client applications.

Example MongoDB Direct Query

```
    select x.* from TABLE(call native('city;{$match:{"city":"FREEDOM"}}')) t,
        xmltable('/city' PASSING JSONTOXML('city', cast(array_get(t.tuple, 1) as BLOB)) COLUMNS
city string, state string) x
```

In the above example, a collection called "city" is looked up with filter that matches the "city" name with "FREEDOM", using "native" procedure and then using the nested tables feature the output is passed to a XMLTABLE construct, where the output from the procedure is sent to a JSONTOXML function to construct a XML then the results of that are exposed in tabular form.

The direct query MUST be in the format

```
"collectionName;{$pipeline instr}+"
```

From Data Virtualization 8.10, MongoDB translator also allows to execute Shell type java script commands like remove, drop, createIndex. For this the command needs to be in format

```
"$ShellCmd;collectionName;operationName;{$instr}+"
```

and example looks like

```
"$ShellCmd;MyTable;remove;{ qty: { $gt: 20 }}"
```

9.9. ODATA TRANSLATOR

The OData translator, known by the type name *"odata"* exposes the OData V2 and V3 data sources and uses the Data Virtualization web services resource adapter for making web service calls. This translator is an extension of the *Web services translator*.

What is OData?

The Open Data Protocol (OData) web protocol is for querying and updating data that provides a way to unlock your data and free it from silos that exist in applications today. OData does this by applying and building upon Web technologies such as HTTP, Atom Publishing Protocol (AtomPub) and JSON to provide access to information from a variety of applications, services, and stores. OData is being used to expose and access information from a variety of sources including, but not limited to, relational databases, file systems, content management systems and traditional Web sites.

Using this specification from the OASIS group, with help from the OData4J framework, Data Virtualization maps OData entities into relational schema. Data Virtualization can read CSDL

(Conceptual Schema Definition Language) from a provided OData endpoint, and convert the OData schema into a relational schema. The following table shows the mapping selections in the OData translator from a CSDL document.

OData	Mapped to relational entity
EntitySet	Table
FunctionImport	Procedure
AssociationSet	Foreign keys on the table*
ComplexType	ignored**

- A many-to-many association will result in a link table that can not be selected from, but can be used for join purposes.
  - When used in functions, an implicit table is exposed. When used to define a embedded table, all the columns will be in-lined.

All CRUD operations will be appropriately mapped to the resulting entity based on the SQL submitted to the OData translator.

1. Usage

Usage of a OData source is similar to that of a JDBC translator. The metadata import is provided through the translator, once the metadata is imported from the source system and exposed in relational terms, then this source can be queried as if the EntitySets and Function Imports were local to the Data Virtualization system.

Table 9.6. Execution properties

Name	Description	Default
DatabaseTimeZone	The time zone of the database. Used when fetchings date, time, or timestamp values.	The system default time zone
SupportsOdataCount	Enables the use of the **$count** option in system queries.	true
SupportsOdataFilter	Enables the use of the **$filter** option in system queries.	true
SupportsOdataOrderBy	Enables the use of the **$orderby** option in system queries.	true
SupportsOdataSkip	Enables the use of the **$skip** option in system queries.	true

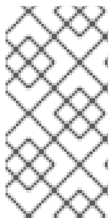
Name	Description	Default
SupportsOdataTop	Enables the use of the **$top** option in system queries.	true

Table 9.7. Importer Properties

Name	Description	Default
schemaNamespace	Namespace of the schema to import.	null
entityContainer	Entity Container Name to import.	default container

Example: Importer settings to import only tables and views from NetflixCatalog

```
<property name="importer.schemaNamespace" value="System.Data.Objects"/>
<property name="importer.entityContainer" value="NetflixCatalog"/>
```



ODATA SERVER IS NOT FULLY COMPATIBLE

The OData server that you connect to might not fully implement the entire OData specification. If the server's OData implementation does not support a feature, set "execution properties" to turn off the corresponding capability, so that Data Virtualization will not push down invalid queries to the translator.

For example, to turn off **$filter**, add the following statement to the virtual database DDL:

```
CREATE SERVER odata FOREIGN DATA WRAPPER "odata-override" OPTIONS
("SupportOdataFilter" 'false');
```

NATIVE QUERIES

The OData translator cannot perform native or direct query execution. However, you can use the *invokehttp* method of the Web services translator to issue REST-based calls, and then use SQLXML to parse results.

USING ODATA AS SERVER.

Data Virtualization can not only consume OData-based data sources, but it can also expose any data source as an OData-based web service.

For more information about configuring an OData server, see *OData support* in the Client Developer's Guide.

9.10. ODATA V4 TRANSLATOR

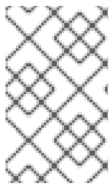
The OData V4 translator, known by the type name *"odata4"* exposes the OData Version 4 data sources

and uses the Data Virtualization web services resource adapter for making web service calls. This translator is extension of *Web Services Translator*. The OData V4 translator is not for use with older OData V1-3 sources. Use the OData translator (*"odata"*) for older OData sources.

What is OData

The Open Data Protocol (OData) Web protocol is for querying and updating data that provides a way to unlock your data and free it from silos that exist in applications today. OData does this by applying and building upon Web technologies such as HTTP, Atom Publishing Protocol (AtomPub), and JSON to provide access to information from a variety of applications, services, and stores. OData is being used to expose and access information from a variety of sources including, but not limited to, relational databases, file systems, content management systems and traditional Web sites.

Using this specification from the OASIS group, with the help from the Olingo framework, Data Virtualization maps OData V4 CSDL (Conceptual Schema Definition Language) document from the OData endpoint provided and converts the OData metadata into Data Virtualization's relational schema. The following table shows the mapping selections in the OData V4 translator from a CSDL document



USING ODATA AS A SERVER

Data Virtualization can not only consume OData-based data sources, but it can expose any data source as an OData based web service. For more information see *OData Support* in the Client Developer's Guide.

OData	Mapped to relational entity
EntitySet	Table
EntityType	Table see [1]
ComplexType	Table see [2]
FunctionImport	Procedure [3]
ActionImport	Procedure [3]
NavigationProperties	Table [4]

[1] Only if the EntityType is exposed as the EntitySet in the Entity container. [2] Only if the complex type is used as property in the exposed EntitySet. This table will be designed as child table with foreign key [1-to-1] or [1-to-many] relationship to the parent.
[3] If the return type is EntityType or ComplexType, the procedure is designed to return a table. [4] Navigation properties are exposed as tables. The table will be created with foreign key relationship to the parent.

All CRUD operations will be appropriately mapped to the resulting entity based on the SQL submitted to the OData translator.

Usage

Usage of a OData source is similar a JDBC translator. The metadata import is supported through the translator, once the metadata is imported from source system and exposed in relational terms, then this source can be queried as if the EntitySets, Function Imports and Action Imports were local to the Data

Virtualization system.

It is not recommended to define your own metadata using Data Virtualization DDL for complex services. There are several extension metadata properties required to enable proper functioning. On non-string properties, a **NATIVE_TYPE** property is expected and should specify the full EDM type name - **Edm.xxx**.

The below is sample VDB that can read metadata service from TripPin service on http://odata.org site.

```
<vdb name="trippin" version="1">
   <model name="trippin">
      <source name="odata4" translator-name="odata4" connection-jndi-name="java:/tripDS"/>
   </model>
</vdb>
```

You can connect to the VDB deployed using Data Virtualization JDBC driver and issue SQL statements like

```
SELECT * FROM trippin.People;
SELECT * FROM trippin.People WHERE UserName = 'russelwhyte';
SELECT * FROM trippin.People p INNER JOIN trippin.People_Friends pf ON p.UserName =
pf.People_UserName; (note that People_UserName is implicitly added by Data Virtualization
metadata)
EXEC GetNearestAirport(lat, lon) ;
```

Execution properties

Sometimes default properties need to adjusted for proper execution of the translator. The following execution properties extend or limit the functionality of the translator based on the physical source capabilities.

Name	Description	Default
SupportsOdataCount	Supports $count	true
SupportsOdataFilter	Supports $filter	true
SupportsOdataOrderBy	Supports $orderby	true
SupportsOdataSkip	Supports $skip	true
SupportsOdataTop	Supports $top	true
SupportsUpdates	Supports INSERT/UPDATE/DELETE	true

The OData server that you connect to might not fully implement the entire OData specification. If the server’s OData implementation does not support a feature, set "execution properties" to turn off the corresponding capability, so that Data Virtualization does not push down invalid queries to the translator.

```
<translator name="odata-override" type="odata">
    <property name="SupportsOdataFilter" value="false"/>
</translator>
```

then use "odata-override" as the translator name on your source model.

Importer properties

The following table lists the importer properties that define the behavior of the translator during metadata import from the physical source.

Name	Description	Default
schemaNamespace	Namespace of the schema to import	null

Example importer settings to only import tables and views from Trippin service exposed on odata.org

```
<property name="importer.schemaNamespace"
value="Microsoft.OData.SampleService.Models.TripPin"/>
```

You can leave this property undefined. If the translator does not detect a configured instance of the property, it specifies the default name of the EntityContainer.

TIP

Native queries - Native or direct query execution is not supported through the OData translator. However, you can use the *invokehttp* method of the Web services translator to issue REST-based calls, and then use SQLXML to parse results.

9.11. OPENAPI TRANSLATOR

The OpenAPI translator, known by the type name *"openapi"* exposes OpenAPI data sources via relational concepts and uses the Data Virtualization WS resource adapter for making web service calls.

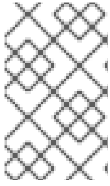
What is OpenAPI?

[OpenAPI is a simple yet powerful representation of your RESTful API. With the largest ecosystem of API tooling on the planet, thousands of developers are supporting OpenAPI in almost every modern programming language and deployment environment. With an OpenAPI-enabled API, you get interactive documentation, client SDK generation, and discoverability.

This translator is compatible with OpenAPI/Swagger v2 and OpenAPI v3.

Usage

Usage of a OpenAPI source is similar any other translator in Data Virtualization. The translator enables metadata import. The metadata is imported from source system’s metadata file and then exposed as stored procedures in Data Virtualization. The source system can be queried by executing these stored procedures in Data Virtualization system.



NOTE

Although parameter order is guaranteed by the Swagger libraries, if you rely upon the native import, it is best if you call procedures using named, rather than positional parameters.

The below is sample VDB that can read metadata from Petstore reference service on http://petstore.swagger.io/ site.

```
<vdb name="petstore" version="1">
    <model visible="true" name="m">
        <property name="importer.metadataUrl" value="/swagger.json"/>
        <source name="s" translator-name="openapi" connection-jndi-name="java:/openapi"/>
    </model>
</vdb>
```

The required resource-adapter configuration will look like

```
<resource-adapter id="openapi">
    <module slot="main" id="org.jboss.teiid.resource-adapter.webservice"/>
    <transaction-support>NoTransaction</transaction-support>
    <connection-definitions>
        <connection-definition class-
name="org.teiid.resource.adapter.ws.WSManagedConnectionFactory" jndi-name="java:/openapi"
enabled="true" use-java-context="true" pool-name="teiid-openapi-ds">
            <config-property name="EndPoint">
                http://petstore.swagger.io/v2
            </config-property>
        </connection-definition>
    </connection-definitions>
</resource-adapter>
```

After you configure the preceding resource-adapter and deploy the VDB successfully, then you can connect to the VDB deployed using Data Virtualization JDBC driver and issue SQL statements such as the following:

```
EXEC findPetsByStatus(('sold',))
EXEC getPetById(1461159803)
EXEC deletePet('', 1461159803)
```

Execution properties

Execution properties extend/limit the functionality of the translator based on the physical source capabilities. Sometimes default properties must be adjusted for proper execution of the translator.

Execution properties

None.

Importer properties

The following table lists the importer properties that define the behavior of the translator during the import of from the physical source.

Name	Description	Default
metadataUrl	URL from which to obtain the OpenAPI metadata. May be a local file using a file: URL.	true
server	The server to use. Otherwise the first server listed will be used.	null
preferredProduces	Preferred Accept MIME type header, this should be one of the OpenAPI 'produces' types;	application/json
preferredConsumes	Preferred Content-Type MIME type header, this should be one of the OpenAPI 'consumer' types;	application/json

TIP

Native queries - The OpenAPI translator cannot perform native or direct query execution. However, you can use the *invokehttp* method of the Web services translator to issue REST-based calls, and then use SQLXML to parse results.

Limitations

The OpenAPI translator does not fully implement all of the features of OpenAPI. The following limitations apply:

- You cannot set the MIME type to **application/xml** in either the **Accept** or **Content-Type** headers.
- File and Map properties cannot be used. As a result, any multi-part payloads are not supported.
- The translator does not process security metadata.
- The translator does not process custom properties that start with **x-**.
- The translator does not work with following JSON schema keywords:
  - **allOf**
  - **multipleOf**
  - **items**

9.12. SALESFORCE TRANSLATORS

You can use the Salesforce translator to run **SELECT**, **DELETE**, **INSERT**, **UPSERT,** and **UPDATE** operations against a Salesforce.com account.

salesforce

The translator, known by the type name ***salesforce***, works with Salesforce API 37.0 and later.

Table 9.8. Execution properties

Name	Description	Default
MaxBulkInsertBatchSize	Batch Size to use to insert bulk inserts.	2048
SupportsGroupBy	Enables **GROUP BY** Pushdown. Set to false to have Data Virtualization process group by aggregations, such as those returning more than 2000 rows which error in SOQL.	true

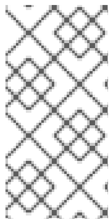
The Salesforce translator can import metadata.

Table 9.9. Import properties

Property Name	Description	Required	Default
NormalizeNames	If the importer should attempt to modify the object/field names so that they can be used unquoted.	false	true
excludeTables	A case-insensitive regular expression that when matched against a table name will exclude it from import. Applied after table names are retrieved. Use a negative look-ahead (?!<inclusion pattern>).* to act as an inclusion filter.	false	n/a
includeTables	A case-insensitive regular expression that when matched against a table name will be included during import. Applied after table names are retrieved from source.	false	n/a
importStatstics	Retrieves cardinalities during import using the REST API explain plan feature.	false	false

Property Name	Description	Required	Default
ModelAuditFields	Add Audit Fields To Model. This includes CreatedXXX, LastModifiedXXX, and SystemModstamp fields.	false	false

NOTE : When both *includeTables* and *excludeTables* patterns are present during the import, the *includeTables* pattern matched first, then the *excludePatterns* will be applied.



NOTE

If you need connectivity to an API version other than what is built in, you may try to use an existing connectivity pair, but in some circumstances - especially accessing a later remote api from an older Java API - this is not possible and results in what appears to be hung connections.

Extension metadata properties

Salesforce is not relational database, however Data Virtualization provides ways to map Saleforce data into relational constructs like Tables and Procedures. You can define a foreign table using DDL in Data Virtualization VDB, which maps to Salesforce's SObject. At runtime, to interpret this table back to a SObject, Data Virtualization decorates or tags this table definition with additional metadata. For example, a table is defined as in the following example:

```
CREATE FOREIGN TABLE Pricebook2 (
 Id string,
 Name string,
 IsActive boolean,
 IsStandard boolean,
 Description string,
 IsDeleted boolean)
 OPTIONS (
   UPDATABLE 'TRUE',
   "teiid_sf:Supports Query" 'TRUE');
```

In the preceding example, the property in the **OPTIONS** clause with the property **"teiid_sf:Supports Query"** set to **TRUE** indicates that you can run **SELECT** commands against this table. The following table lists the metadata extension properties that can be used in a Salesforce schema.

Property Name	Description	Required	Default	Applies To
Supports Query	You can run **SELECT** commands against the table.	false	true	Table

Property Name	Description	Required	Default	Applies To
Supports Retrieve	You can retrieve the results of **SELECT** commands run against the table.	false	true	Table

SQL processing

Salesforce does not provide the same set of functionality as a relational database. For example, Salesforce does not support arbitrary joins between tables. However, working in combination with the Data Virtualization Query Planner, the Salesforce connector can use nearly all of the SQL syntax capabilities in Data Virtualization. The Salesforce Connector executes SQL commands by "pushing down" the command to Salesforce whenever possible, depending on the available capabilities. Data Virtualization will automatically provide additional database functionality when the Salesforce Connector does not explicitly enable use of a given SQL construct. In cases where certain SQL capabilities cannot be pushed down to Salesforce, Data Virtualization will push down the capabilities that it can, and fetch a set of data from Salesforce. Then, Data Virtualization will evaluate the additional capabilities, creating a subset of the original data set. Finally, Data Virtualization will pass the result to the client.

If you issue queries with a **GROUP BY** clause, and you receive a Salesforce error that indicates that **queryMore** is not supported, you can either add limits, or set the execution property **SupportsGroupBy** to **false**.

```
SELECT array_agg(Reports) FROM Supervisor where Division = 'customer support';
```

Neither Salesforce, nor the Salesforce Connector support the **array_agg()** scalar. however, both are compatible with the **CompareCriteriaEquals** query, so the connector transforms the query that it receives into this query to Salesforce.

```
SELECT Reports FROM Supervisor where Division = 'customer support';
```

The array_agg() function will be applied by the Data Virtualization Query Engine to the result set returned by the connector.

In some cases, multiple calls to the Salesforce application will be made to process the SQL that is passed to the connector.

```
DELETE From Case WHERE Status = 'Closed';
```

The API in Salesforce to delete objects can delete by object ID only. In order to accomplish this, the Salesforce connector will first execute a query to get the IDs of the correct objects, and then delete those objects. So the above DELETE command will result in the following two commands.

```
SELECT ID From Case WHERE Status = 'Closed';
DELETE From Case where ID IN (<result of query>);
```

NOTE： The Salesforce API DELETE call is not expressed in SQL, but the above is an equivalent SQL expression.

It's useful to be aware of incompatible capabilities, in order to avoid fetching large data sets from Salesforce and making you queries as performant as possible. For information about the SQL constructs that you can push down to Salesforce, see Compatible SQL capabilities.

Selecting from multi-select picklists

A multi-select picklist is a field type in Salesforce that can contain multiple values in a single field. Query criteria operators for fields of this type in SOQL are limited to EQ, NE, includes and excludes. For the Salesforce documentation about how to select from multi-select picklists, see Querying Multi-select Picklists

Data Virtualization SQL does not support the includes or excludes operators, but the Salesforce connector provides user-defined function definitions for these operators that provide equivalent functionality for fields of type multi-select. The definition for the functions is:

```
boolean includes(Column column, String param)
boolean excludes(Column column, String param)
```

For example, take a single multi-select picklist column called Status that contains all of these values.

- current
- working
- critical

For that column, all of the below are valid queries:

```
SELECT * FROM Issue WHERE true = includes (Status, 'current, working' );
SELECT * FROM Issue WHERE true = excludes (Status, 'current, working' );
SELECT * FROM Issue WHERE true = includes (Status, 'current;working, critical' );
```

EQ and NE criteria will pass to Salesforce as supplied. For example, these queries will not be modified by the connector.

```
SELECT * FROM Issue WHERE Status = 'current';
SELECT * FROM Issue WHERE Status = 'current;critical';
SELECT * FROM Issue WHERE Status != 'current;working';
```

Selecting all objects

You can use the Salesforce connector to call the **queryAll** operation from the Salesforce API. The **queryAll** operation is equivalent to the query operation with the exception that it returns data about all current and deletedobjects in the system.

The connector determines if it will call the query or **queryAll** operation via reference to the **isDeleted** property present on each Salesforce object, and modeled as a column on each table generated by the importer. By default this value is set to **false** when the model is generated and thus the connector calls query. Users are free to change the value in the model to **true**, changing the default behavior of the connector to be **queryAll**.

The behavior is different if **isDeleted** is used as a parameter in the query. If the **isDeleted** column is used as a parameter in the query, and the value is **true**, then the connector calls **queryAll**.

```
select * from Contact where isDeleted = true;
```

If the **isDeleted** column is used as a parameter in the query, and the value is **false**, then the connector that performs the default behavior will call the query.

```
select * from Contact where isDeleted = false;
```

Selecting updated objects

If the option is selected when importing metadata from Salesforce, a GetUpdated procedure is generated in the model with the following structure:

```
GetUpdated (ObjectName IN string,
    StartDate IN datetime,
    EndDate IN datetime,
    LatestDateCovered OUT datetime)
returns
    ID string
```

See the description of the GetUpdated operation in the Salesforce documentation for usage details.

Selecting deleted objects

If the option is selected when importing metadata from Salesforce, a GetDeleted procedure is generated in the model with the following structure:

```
GetDeleted (ObjectName IN string,
    StartDate IN datetime,
    EndDate IN datetime,
    EarliestDateAvailable OUT datetime,
    LatestDateCovered OUT datetime)
returns
    ID string,
    DeletedDate datetime
```

See the description of the GetDeleted operation in the Salesforce documentation for usage details.

Relationship queries

Unlike a relational database, Salesforce does not support join operations, but it does have support for queries that include parent-to-child or child-to-parent relationships between objects. These are termed Relationship Queries. You can run Relationship Queries in the SalesForce connector through Outer Join syntax.

```
SELECT Account.name, Contact.Name from Contact LEFT OUTER JOIN Account
on Contact.Accountid = Account.id
```

This query shows the correct syntax to query a SalesForce model with to produce a relationship query from child to parent. It resolves to the following query to SalesForce.

```
SELECT Contact.Account.Name, Contact.Name FROM Contact
```

```
select Contact.Name, Account.Name from Account Left outer Join Contact
on Contact.Accountid = Account.id
```

This query shows the correct syntax to query a SalesForce model with to produce a relationship query from parent to child. It resolves to the following query to SalesForce.

```
SELECT Account.Name, (SELECT Contact.Name FROM
Account.Contacts) FROM Account
```

See the description of the Relationship Queries operation in the SalesForce documentation for limitations.

Bulk insert queries

You can also use bulk insert statements in the SalesForce translator by using JDBC batch semantics or SELECT INTO semantics. The batch size is determined by the execution property *MaxBulkInsertBatchSize*, which can be overridden in the vdb file. The default value of the batch is 2048. The bulk insert feature uses the async REST based API exposed by Salesforce for execution for better performance.

Bulk selects

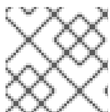
When querying tables with more than 10,000,000 records, or if experiencing timeouts with just result batching, Data Virtualization can issue queries to Salesforce using the bulk API. When using a bulk select, primary key (PK) chunking is enabled if it is compatible with the query.

The use of the bulk api requires a source hint in the query:

```
SELECT /*+ sh salesforce:'bulk' */ Name ... FROM Account
```

Where salesforce is the source name of the target source.

The default chunk size of 100,000 records will be used.



NOTE

This feature is only supported in the Salesforce API version 28 or higher.

Compatible SQL capabilities

You ca use the following SQL capabilities with the Salesforce Connector. These SQL constructs will be pushed down to Salesforce.

- SELECT command
- INSERT Command
- UPDATE Command
- DELETE Command
- NotCriteria
- OrCriteria
- CompareCriteriaEquals
- CompareCriteriaOrdered

- IsNullCritiera
- InCriteria
- LikeCriteria - Can be used for String fields only.
- RowLimit
- Basic Aggregates
- OuterJoins with join criteria KEY

Native Queries

Salesforce procedures may optionally have native queries associated with them. For more information, see *Parameterizable native queries* in Translators. The operation prefix (select;, insert;, update;, delete; - see below for more) must be present in the native-query, but it will not be issued as part of the query to the source.

Example DDL for a Salesforce native procedure

```
CREATE FOREIGN PROCEDURE proc (arg1 integer, arg2 string) OPTIONS ("teiid_rel:native-query"
'search;SELECT ... complex SOQL ... WHERE col1 = $1 and col2 = $2')
returns (col1 string, col2 string, col3 timestamp);
```

Direct query procedure

This feature is turned off by default because of the security risk this exposes to execute any command against the source. To enable direct query procedures, set the execution property called **SupportsDirectQueryProcedure** to **true**. For more information, see *Override the execution properties* in Chapter 9, *Translators*.

TIP

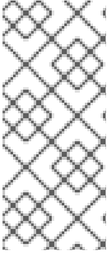
By default the name of the procedure that executes the queries directly is called **native**. For information about how to change the default name, see *Override the execution properties* in Chapter 9, *Translators*.

The Salesforce translator provides a procedure to execute any ad-hoc SOQL query directly against the source without Data Virtualization parsing or resolving. Since the metadata of this procedure's results are not known to Data Virtualization, they are returned as an object array. ARRAYTABLE can be used construct tabular output for consumption by client applications. Data Virtualization exposes this procedure with a simple query structure as follows:

Select example

```
SELECT x.* FROM (call sf_source.native('search;SELECT Account.Id, Account.Type, Account.Name
FROM Account')) w,
 ARRAYTABLE(w.tuple COLUMNS "id" string , "type" string, "name" String) AS x
```

from the above code, the "search" keyword followed by a query statement.



NOTE

The SOQL is treated as a parameterized native query so that parameter values may be inserted in the query string properly. For more information, see *Parameterizable native queries* in Translators. The results returned by search may contain the object Id as the first column value regardless of whether it was selected. Also queries that select columns from multiple object types will not be correct.

Delete Example

```
SELECT x.* FROM (call sf_source.native('delete;', 'id1', 'id2')) w,
 ARRAYTABLE(w.tuple COLUMNS "updatecount" integer) AS x
```

form the above code, the "delete;" keyword followed by the ids to delete as varargs.

Create example

```
SELECT x.* FROM
 (call sf_source.native('create;type=table;attributes=one,two,three', 'one', 2, 3.0)) w,
 ARRAYTABLE(w.tuple COLUMNS "update_count" integer) AS x
```

form the above code, the "create" or "update" keyword must be followed by the following properties. Attributes must be matched positionally by the procedure variables - thus in the example attribute two will be set to 2.

Property Name	Description	Required
type	Table Name	Yes
attributes	comma separated list of names of the columns	no

The values for each attribute is specified as separate argument to the "native" procedure.

Update is similar to create, with one more extra property called "id", which defines identifier for the record.

Update example

```
SELECT x.* FROM
 (call sf_source.native('update;id=pk;type=table;attributes=one,two,three', 'one', 2, 3.0)) w,
 ARRAYTABLE(w.tuple COLUMNS "update_count" integer) AS x
```

TIP

By default the name of the procedure that executes the queries directly is called native, however you can add set an override execution property in the DDL file to change it.

9.13. REST TRANSLATOR

The Rest translator, known by the type name *rest*, exposes stored procedures for calling REST services. Results from this translator will commonly be used with the TEXTTABLE, JSONTABLE, or XMLTABLE table functions to use CSV, JSON, or XML formated data.

Execution properties

There are no *rest* importer settings, but it can provide metadata for VDBs.

Usage

The *rest* translator exposes low level procedures for accessing web services.

InvokeHTTP procedure

invokeHttp can return the byte contents of an HTTP(S) call.

```
Procedure invokeHttp(action in STRING, request in OBJECT, endpoint in STRING, stream in BOOLEAN, contentType out STRING, headers in CLOB) returns BLOB
```

Action indicates the HTTP method (GET, POST, etc.), which defaults to POST.

A null value for endpoint will use the default value. The default endpoint is specified in the *rest* source configuration. The endpoint URL may be absolute or relative. If it's relative then it will be combined with the default endpoint.

Since multiple parameters are not required to have values, it is often more clear to call the invokeHttp procedure with named parameter syntax.

```
call invokeHttp(action=>'GET')
```

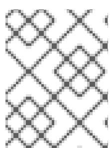
The request can be one of SQLXML, STRING, BLOB, or CLOB. The request will be sent as the POST payload in byte form. For STRING/CLOB values this will default to the UTF-8 encoding. To control the byte encoding, see the to_bytes function.

The optional headers parameter can be used to specify the request header values as a JSON value. The JSON value should be a JSON object with primitive or list of primitive values.

```
call invokeHttp(... headers=>jsonObject('application/json' as "Content-Type", jsonArray('gzip', 'deflate') as "Accept-Encoding"))
```

Recommendations for setting headers parameter:

- **Content-Type** might be necessary if the HTTP POST/PUT method is invoked.
- *Accept* is necessary if you want to control return Media Type.



NATIVE QUERIES

You cannot use native queries or direct query execution procedures with the web services translator.

Streaming considerations

If the stream parameter is set to **true**, then the resulting LOB value may only be used a single time. If stream is **null** or **false**, then the engine may need to save a copy of the result for repeated use. Care must be used as some operations, such as casting or **XMLPARSE** might perform validation which results

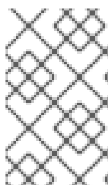
in the stream being consumed.

9.14. WEB SERVICES TRANSLATOR

The Web Services translator, known by the type name *soap* or *ws*, exposes stored procedures for calling web/SOAP services. Results from this translator will commonly be used with the TEXTTABLE or XMLTABLE table functions to use CSV or XML formated data.

Execution properties

Name	Description	When Used	Default
DefaultBinding	The binding that should be used if one is not specified. Can be one of HTTP, SOAP11, or SOAP12.	invoke*	SOAP12
DefaultServiceMode	The default service mode. For SOAP, MESSAGE mode indicates that the request will contain the entire SOAP envelope. and not just the contents of the SOAP body. Can be one of MESSAGE or PAYLOAD	invoke* or WSDL call	PAYLOAD
XMLParamName	Used with the HTTP binding (typically with the GET method) to indicate that the request document should be part of the query string.	invoke*	null - unused



NOTE

Setting the proper binding value on the translator is recommended as it removes the need for callers to pass an explicit value. If your service is actually uses SOAP11, but the binding used SOAP12 you will receive execution failures.

There are no importer settings, but it can provide metadata for VDBs. If the connection is configured to point at a specific WSDL, the translator will import all SOAP operations under the specified service and port as procedures.

Importer properties

When specifying the importer property, it must be prefixed with "importer.". Example: importer.tableTypes

Name	Description	Default
importWSDL	Import the metadata from the WSDL URL configured in resource-adapter.	true

Usage

The translator exposes low level procedures for accessing web services.

Invoke procedure

Invoke allows for multiple binding, or protocol modes, including HTTP, SOAP11, and SOAP12.

```
Procedure invoke(binding in STRING, action in STRING, request in XML, endpoint in STRING, stream in BOOLEAN) returns XML
```

The binding may be one of null (to use the default) HTTP, SOAP11, or SOAP12. Action with a SOAP binding indicates the SOAPAction value. Action with a HTTP binding indicates the HTTP method (GET, POST, etc.), which defaults to POST.

A null value for the binding or endpoint will use the default value. The default endpoint is specified in the source configuration. The endpoint URL may be absolute or relative. If it's relative then it will be combined with the default endpoint.

Since multiple parameters are not required to have values, it is often more clear to call the invoke procedure with named parameter syntax.

```
call invoke(binding=>'HTTP', action=>'GET')
```

The request XML should be a valid XML document or root element.

InvokeHTTP procedure

invokeHttp can return the byte contents of an HTTP(S) call.

```
Procedure invokeHttp(action in STRING, request in OBJECT, endpoint in STRING, stream in BOOLEAN, contentType out STRING, headers in CLOB) returns BLOB
```

Action indicates the HTTP method (GET, POST, etc.), which defaults to POST.

A null value for endpoint will use the default value. The default endpoint is specified in the source configuration. The endpoint URL may be absolute or relative. If it's relative then it will be combined with the default endpoint.

Since multiple parameters are not required to have values, it is often more clear to call the invokeHttp procedure with named parameter syntax.

```
call invokeHttp(action=>'GET')
```

The request can be one of SQLXML, STRING, BLOB, or CLOB. The request will be sent as the POST payload in byte form. For STRING/CLOB values this will default to the UTF-8 encoding. To control the byte encoding, see the to_bytes function.

The optional headers parameter can be used to specify the request header values as a JSON value. The JSON value should be a JSON object with primitive or list of primitive values.

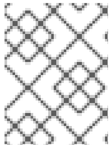
```
call invokeHttp(... headers=>jsonObject('application/json' as "Content-Type", jsonArray('gzip', 'deflate') as "Accept-Encoding"))
```

Recommendations for setting headers parameter:

- **Content-Type** might be necessary if the HTTP POST/PUT method is invoked.
- *Accept* is necessary if you want to control return Media Type.

WSDL based procedures

The procedures above give you anonymous way to execute any web service methods by supplying an endpoint, with this mechanism you can alter the endpoint defined in WSDL with a different endpoint. However, if you have access to the WSDL, then you can configure the WSDL URL in the web-service resource-adapter's connection configuration, Web Service translator can parse the WSDL and provide the methods under configured port as pre-built procedures as its metadata. If you are using the default native metadata import, you will see the procedures in your web service's source model.



NATIVE QUERIES

You cannot use native queries or direct query execution procedures with the web services translator.

Streaming considerations

If the stream parameter is set to **true**, then the resulting LOB value may only be used a single time. If stream is **null** or **false**, then the engine may need to save a copy of the result for repeated use. Care must be used as some operations, such as casting or **XMLPARSE** might perform validation which results in the stream being consumed.

CHAPTER 10. FEDERATED PLANNING

At its core, Data Virtualization is a federated, relational query engine. This query engine allows you to treat all of your data sources as one virtual database, and access them through a single SQL query. As a result, instead of focusing on hand-coding joins, you can focus on building your application, and on running other relational operations between data sources.

10.1. PLANNING OVERVIEW

When the query engine receives an incoming SQL query it performs the following operations:

1. **Parsing** – Validates syntax and convert to internal form.
2. **Resolving** – Links all identifiers to metadata and functions to the function library.
3. **Validating** – Validates SQL semantics based on metadata references and type signatures.
4. **Rewriting** – Rewrites SQL to simplify expressions and criteria.
5. **Logical plan optimization** – Converts the rewritten canonical SQL to a logical plan for in-depth optimization. The Data Virtualization optimizer is predominantly rule-based. Based upon the query structure and hints, a certain rule set will be applied. These rules may trigger in turn trigger the execution of more rules. Within several rules, Data Virtualization also takes advantage of costing information. The logical plan optimization steps can be seen by using the `` `SET SHOWPLAN DEBUG` `` clause, as described in the Client Development Guide. For sample steps, see *Reading a debug plan* in Query Planner. For more information about logical plan nodes and rule-based optimization, see Query Planner.
6. **Processing plan conversion** – Converts the logic plan to an executable form where the nodes represent basic processing operations. The final processing plan is displayed as a query plan. For more information, see Query plans.

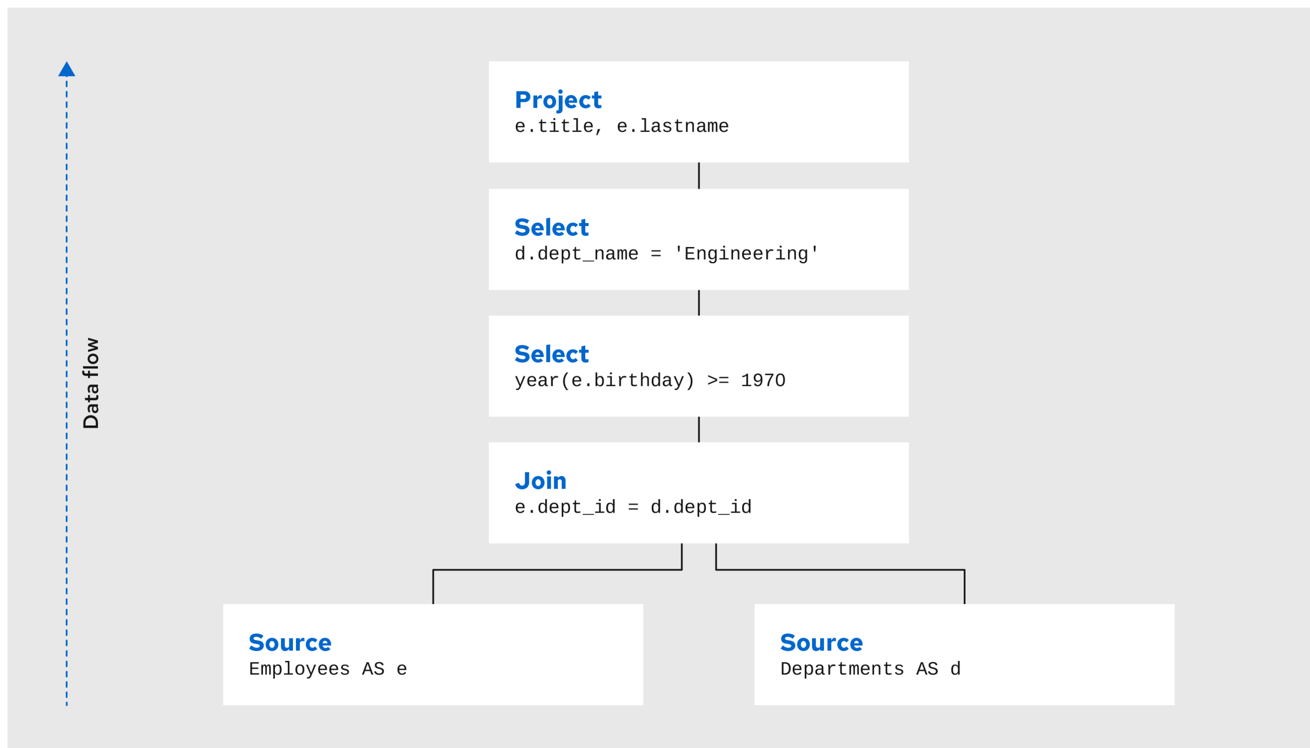
The logical query plan is a tree of operations that is used to transform data in source tables to the expected result set. In the tree, data flows from the bottom (tables) to the top (output). The primary logical operations are *select* (select or filter rows based on a criteria), *project* (project or compute column values), *join*, *source* (retrieve data from a table), *sort* (ORDER BY), *duplicate removal* (SELECT DISTINCT), *group* (GROUP BY), and *union* (UNION).

For example, consider the following query that retrieves all engineering employees born since 1970.

Example query

```sql
SELECT e.title, e.lastname FROM Employees AS e JOIN Departments AS d ON e.dept_id =
d.dept_id WHERE year(e.birthday) >= 1970 AND d.dept_name = 'Engineering'
```

Logically, the data from the Employees and Departments tables are retrieved, then joined, then filtered as specified, and finally the output columns are projected. The canonical query plan thus looks like this:



Data flows from the tables at the bottom upwards through the join, through the select, and finally through the project to produce the final results. The data passed between each node is logically a result set with columns and rows.

Of course, this is what happens *logically* – it is not how the plan is actually executed. Starting from this initial plan, the query planner performs transformations on the query plan tree to produce an equivalent plan that retrieves the same results faster. Both a federated query planner and a relational database planner deal with the same concepts and many of the same plan transformations. In this example, the criteria on the Departments and Employees tables will be pushed down the tree to filter the results as early as possible.

In both cases, the goal is to retrieve the query results in the fastest possible time. However, the relational database planner achieve this primarily by optimizing the access paths in pulling data from storage.

In contrast, a federated query planner is less concerned about storage access, because it is typically pushing that burden to the data source. The most important consideration for a federated query planner is minimizing data transfer.

10.2. QUERY PLANNER

For each sub-command in the user command an appropriate kind of sub-planner is used (relational, XML, procedure, etc).

Each planner has three primary phases:

1. Generate canonical plan

2. Optimization

3. Plan to process converter – Converts plan data structure into a processing form.

Relational planner

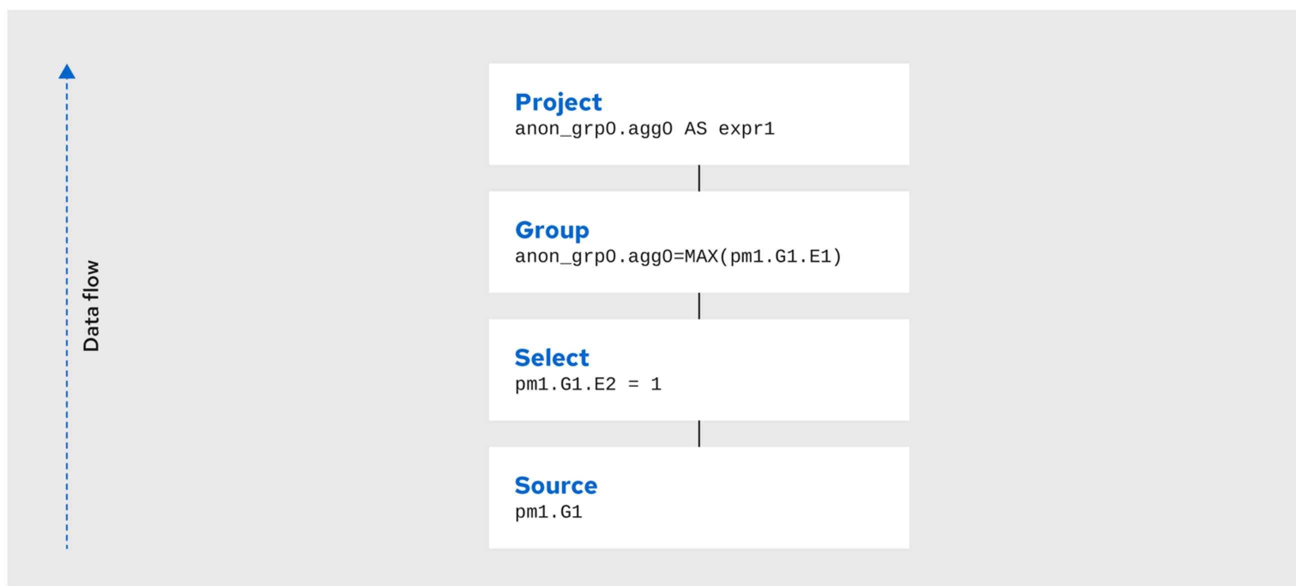
A relational processing plan is created by the optimizer after the logical plan is manipulated by a series of rules. The application of rules is determined both by the query structure and by the rules themselves. The node structure of the debug plan resembles that of the processing plan, but the node types more logically represent SQL operations.

Canonical plan and all nodes

As described in the Planning overview, a SQL statement submitted to the query engine is parsed, resolved, validated, and rewritten before it is converted into a canonical plan form. The canonical plan form most closely resembles the initial SQL structure. A SQL select query has the following possible clauses (all but SELECT are optional): WITH, SELECT, FROM, WHERE, GROUP BY, HAVING, ORDER BY, LIMIT. These clauses are logically executed in the following order:

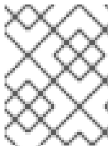
1. WITH (create common table expressions) – Processed by a specialized PROJECT NODE.
2. FROM (read and join all data from tables) – Processed by a SOURCE node for each from clause item, or a Join node (if >1 table).
3. WHERE (filter rows) – Processed by a SELECT node.
4. GROUP BY (group rows into collapsed rows) – Processed by a GROUP node.
5. HAVING (filter grouped rows) – Processed by a SELECT node.
6. SELECT (evaluate expressions and return only requested rows) – Processed by a PROJECT node and DUP_REMOVE node (for SELECT DISTINCT).
7. INTO – Processed by a specialized PROJECT with a SOURCE child.
8. ORDER BY (sort rows) – Processed by a SORT node.
9. LIMIT (limit result set to a certain range of results) – Processed by a LIMIT node.

For example, a SQL statement such as **SELECT max(pm1.g1.e1) FROM pm1.g1 WHERE e2 = 1** creates a logical plan:



```
Project(groups=[anon_grp0], props={PROJECT_COLS=[anon_grp0.agg0 AS expr1]})
  Group(groups=[anon_grp0], props={SYMBOL_MAP={anon_grp0.agg0=MAX(pm1.G1.E1)}})
    Select(groups=[pm1.G1], props={SELECT_CRITERIA=pm1.G1.E2 = 1})
      Source(groups=[pm1.G1])
```

Here the Source corresponds to the FROM clause, the Select corresponds to the WHERE clause, the Group corresponds to the implied grouping to create the max aggregate, and the Project corresponds to the SELECT clause.



NOTE

The effect of grouping generates what is effectively an inline view, **anon_grp0**, to handle the projection of values created by the grouping.

Table 10.1. Node Types

Type Name	Description
ACCESS	A source access or plan execution.
DUP_REMOVE	Removes duplicate rows
JOIN	A join (LEFT OUTER, FULL OUTER, INNER, CROSS, SEMI, and so forth).
PROJECT	A projection of tuple values
SELECT	A filtering of tuples
SORT	An ordering operation, which may be inserted to process other operations such as joins.
SOURCE	Any logical source of tuples including an inline view, a source access, XMLTABLE, and so forth.
GROUP	A grouping operation.
SET_OP	A set operation (UNION/INTERSECT/EXCEPT).
NULL	A source of no tuples.
TUPLE_LIMIT	Row offset / limit

Node properties

Each node has a set of applicable properties that are typically shown on the node.

Table 10.2. Access properties

Property Name	Description
ATOMIC_REQUEST	The final form of a source request.
MODEL_ID	The metadata object for the target model/schema.
PROCEDURE_CRITERIA/PROCEDURE_INPUTS/PROCEDURE_DEFAULTS	Used in planning procedureal relational queries.
IS_MULTI_SOURCE	set to true when the node represents a multi-source access.
SOURCE_NAME	used to track the multi-source source name.
CONFORMED_SOURCES	tracks the set of conformed sources when the conformed extension metadata is used.
SUB_PLAN/SUB_PLANS	used in multi-source planning.

Table 10.3. Set operation properties

Property Name	Description
SET_OPERATION/USE_ALL	defines the set operation(UNION/INTERSECT/EXCEPT) and if all rows or distinct rows are used.

Table 10.4. Join properties

Property Name	Description
JOIN_CRITERIA	All join predicates.
JOIN_TYPE	Type of join (INNER, LEFT OUTER, and so forth).
JOIN_STRATEGY	The algorithm to use (nested loop, merge, and so forth).
LEFT_EXPRESSIONS	The expressions in equi-join predicates that originate from the left side of the join.
RIGHT_EXPRESSIONS	The expressions in equi-join predicates that originate from the right side of the join.
DEPENDENT_VALUE_SOURCE	set if a dependent join is used.
NON_EQUI_JOIN_CRITERIA	Non-equi join predicates.

Property Name	Description
SORT_LEFT	If the left side needs sorted for join processing.
SORT_RIGHT	If the right side needs sorted for join processing.
IS_OPTIONAL	If the join is optional.
IS_LEFT_DISTINCT	If the left side is distinct with respect to the equi join predicates.
IS_RIGHT_DISTINCT	If the right side is distinct with respect to the equi join predicates.
IS_SEMI_DEP	If the dependent join represents a semi-join.
PRESERVE	If the preserve hint is preserving the join order.

Table 10.5. Project properties

Property Name	Description
PROJECT_COLS	The expressions projected.
INTO_GROUP	The group targeted if this is a select into or insert with a query expression.
HAS_WINDOW_FUNCTIONS	True if window functions are used.
CONSTRAINT	The constraint that must be met if the values are being projected into a group.
UPSERT	If the insert is an upsert.

Table 10.6. Select properties

Property Name	Description
SELECT_CRITERIA	The filter.
IS_HAVING	If the filter is applied after grouping.
IS_PHANTOM	True if the node is marked for removal, but temporarily left in the plan.
IS_TEMPORARY	Inferred criteria that may not be used in the final plan.

Property Name	Description
IS_COPIED	If the criteria has already been processed by rule copy criteria.
IS_PUSHED	If the criteria is pushed as far as possible.
IS_DEPENDENT_SET	If the criteria is the filter of a dependent join.

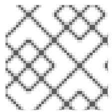
Table 10.7. Sort properties

Property Name	Description
SORT_ORDER	The order by that defines the sort.
UNRELATED_SORT	If the ordering includes a value that is not being projected.
IS_DUP_REMOVAL	If the sort should also perform duplicate removal over the entire projection.

Table 10.8. Source properties

Property Name	Description
SYMBOL_MAP	The mapping from the columns above the source to the projected expressions. Also present on Group nodes.
PARTITION_INFO	The partitioning of the union branches.
VIRTUAL_COMMAND	If the source represents an view or inline view, the query that defined the view.
MAKE_DEP	Hint information.
PROCESSOR_PLAN	The processor plan of a non-relational source(typically from the NESTED_COMMAND).
NESTED_COMMAND	The non-relational command.
TABLE_FUNCTION	The table function (XMLTABLE, OBJECTTABLE, and so forth.) defining the source.
CORRELATED_REFERENCES	The correlated references for the nodes below the source.

Property Name	Description
MAKE_NOT_DEP	If make not dep is set.
INLINE_VIEW	If the source node represents an inline view.
NO_UNNEST	If the no_unnest hint is set.
MAKE_IND	If the make ind hint is set.
SOURCE_HINT	The source hint. See Federated optimizations.
ACCESS_PATTERNS	Access patterns yet to be satisfied.
ACCESS_PATTERN_USED	Satisfied access patterns.
REQUIRED_ACCESS_PATTERN_GROUPS	Groups needed to satisfy the access patterns. Used in join planning.



NOTE

Many source properties also become present on associated access nodes.

Table 10.9. Group properties

Property Name	Description
GROUP_COLS	The grouping columns.
ROLLUP	If the grouping includes a rollup.

Table 10.10. Tuple limit properties

Property Name	Description
MAX_TUPLE_LIMIT	Expression that evaluates to the max number of tuples generated.
OFFSET_TUPLE_COUNT	Expression that evaluates to the tuple offset of the starting tuple.
IS_IMPLICIT_LIMIT	If the limit is created by the rewriter as part of a subquery optimization.
IS_NON_STRICT	If the unordered limit should not be enforced strictly.

Table 10.11. General and costing properties

Property Name	Description
OUTPUT_COLS	The output columns for the node. Is typically set after rule assign output elements.
EST_SET_SIZE	Represents the estimated set size this node would produce for a sibling node as the independent node in a dependent join scenario.
EST_DEP_CARDINALITY	Value that represents the estimated cardinality (amount of rows) produced by this node as the dependent node in a dependent join scenario.
EST_DEP_JOIN_COST	Value that represents the estimated cost of a dependent join (the join strategy for this could be Nested Loop or Merge).
EST_JOIN_COST	Value that represents the estimated cost of a merge join (the join strategy for this could be Nested Loop or Merge).
EST_CARDINALITY	Represents the estimated cardinality (amount of rows) produced by this node.
EST_COL_STATS	Column statistics including number of null values, distinct value count, and so forth.
EST_SELECTIVITY	Represents the selectivity of a criteria node.

Rules

Relational optimization is based upon rule execution that evolves the initial plan into the execution plan. There are a set of pre-defined rules that are dynamically assembled into a rule stack for every query. The rule stack is assembled based on the contents of the user's query and the views/procedures accessed. For example, if there are no view layers, then rule Merge Virtual, which merges view layers together, is not needed and will not be added to the stack. This allows the rule stack to reflect the complexity of the query.

Logically the plan node data structure represents a tree of nodes where the source data comes up from the leaf nodes (typically Access nodes in the final plan), flows up through the tree and produces the user's results out the top. The nodes in the plan structure can have bidirectional links, dynamic properties, and allow any number of child nodes. Processing plans in contrast typically have fixed properties.

Plan rule manipulate the plan tree, fire other rules, and drive the optimization process. Each rule is designed to perform a narrow set of tasks. Some rules can be run multiple times. Some rules require a specific set of precursors to run properly.

- Access Pattern Validation – Ensures that all access patterns have been satisfied.
- Apply Security – Applies row and column level security.

- Assign Output Symbol – This rule walks top down through every node and calculates the output columns for each node. Columns that are not needed are dropped at every node, which is known as projection minimization. This is done by keeping track of both the columns needed to feed the parent node and also keeping track of columns that are "created" at a certain node.

- Calculate Cost – Adds costing information to the plan

- Choose Dependent – This rule looks at each join node and determines whether the join should be made dependent and in which direction. Cardinality, the number of distinct values, and primary key information are used in several formulas to determine whether a dependent join is likely to be worthwhile. The dependent join differs in performance ideally because a fewer number of values will be returned from the dependent side.
  Also, we must consider the number of values passed from independent to dependent side. If that set is larger than the maximum number of values in an IN criteria on the dependent side, then we must break the query into a set of queries and combine their results. Executing each query in the connector has some overhead and that is taken into account. Without costing information a lot of common cases where the only criteria specified is on a non-unique (but strongly limiting) field are missed.

  A join is eligible to be dependent if:

  - There is at least one equi-join criterion, for example, **tablea.col = tableb.col**

  - The join is not a full outer join and the dependent side of the join is on the inner side of the join.

The join will be made dependent if one of the following conditions, listed in precedence order, holds:

- There is an unsatisfied access pattern that can be satisfied with the dependent join criteria.

- The potential dependent side of the join is marked with an option makedep.

- (4.3.2) if costing was enabled, the estimated cost for the dependent join (5.0+ possibly in each direction in the case of inner joins) is computed and compared to not performing the dependent join. If the costs were all determined (which requires all relevant table cardinality, column ndv, and possibly nnv values to be populated) the lowest is chosen.

- If key metadata information indicates that the potential dependent side is not "small" and the other side is "not small" or (5.0.1) the potential dependent side is the inner side of a left outer join.

Dependent join is the key optimization we use to efficiently process multi-source joins. Instead of reading all of source A and all of source B and joining them on A.x = B.x, we read all of A, and then build a set of A.x that are passed as a criteria when querying B. In cases where A is small and B is large, this can drastically reduce the data retrieved from B, thus greatly speeding the overall query.

- Choose Join Strategy – Choose the join strategy based upon the cost and attributes of the join.

- Clean Criteria – Removes phantom criteria.

- Collapse Source – Takes all of the nodes below an access node and creates a SQL query representation.

- Copy Criteria – This rule copies criteria over an equality criteria that is present in the criteria of a join. Since the equality defines an equivalence, this is a valid way to create a new criteria that may limit results on the other side of the join (especially in the case of a multi-source join).

- Decompose Join – This rule performs a partition-wise join optimization on joins of a partitioned union. For more information, see *Partitioned unions* in Federated optimizations. The decision to decompose is based upon detecting that each side of the join is a partitioned union (note that non-ANSI joins of more than 2 tables may cause the optimization to not detect the appropriate join). The rule currently only looks for situations where at most 1 partition matches from each side.

- Implement Join Strategy – Adds necessary sort and other nodes to process the chosen join strategy

- Merge Criteria – Combines select nodes

- Merge Virtual – Removes view and inline view layers

- Place Access – Places access nodes under source nodes. An access node represents the point at which everything below the access node gets pushed to the source or is a plan invocation. Later rules focus on either pushing under the access or pulling the access node up the tree to move more work down to the sources. This rule is also responsible for placing access patterns. For more information, see *Access patterns* in Federated optimizations

- Plan Joins – This rule attempts to find an optimal ordering of the joins performed in the plan, while ensuring that access pattern dependencies are met. This rule has three main steps.

    1. It must determine an ordering of joins that satisfy the access patterns present.

    2. It will heuristically create joins that can be pushed to the source (if a set of joins are pushed to the source, we will not attempt to create an optimal ordering within that set. More than likely it will be sent to the source in the non-ANSI multi-join syntax and will be optimized by the database).

    3. It will use costing information to determine the best left-linear ordering of joins performed in the processing engine. This third step will do an exhaustive search for 7 or less join sources and is heuristically driven by join selectivity for 8 or more sources.

- Plan Outer Joins – Reorders outer joins as permitted to improve push down.

- Plan Procedures – Plans procedures that appear in procedural relational queries.

- Plan Sorts – Optimizations around sorting, such as combining sort operations or moving projection.

- Plan Subqueries – New for Data Virtualization 12. Generalizes the subquery optimization that was performed in Merge Criteria to allow for the creation of join plans from subqueries in both projection and filtering.

- Plan Unions – Reorders union children for more pushdown.

- Plan Aggregates – Performs aggregate decomposition over a join or union.

- Push Limit – Pushes the affect of a limit node further into the plan.

- Push Non-Join Criteria – This rule will push predicates out of an on clause if it is not necessary for the correctness of the join.

- Push Select Criteria – Push select nodes as far as possible through unions, joins, and views layers toward the access nodes. In most cases movement down the tree is good as this will filter rows earlier in the plan. We currently do not undo the decisions made by Push Select Criteria.

However in situations where criteria cannot be evaluated by the source, this can lead to sub-optimal plans.

- Push Large IN – Push **IN** predicates that are larger than the translator can process directly to be processed as a dependent set.

One of the most important optimization related to pushing criteria, is how the criteria will be pushed through join. Consider the following plan tree that represents a subtree of the plan for the query **select * from A inner join b on (A.x = B.x) where B.y = 3**

```
SELECT (B.y = 3)
       |
      JOIN - Inner Join on (A.x = B.x)
     /    \
  SRC (A)   SRC (B)
```

NOTE

SELECT nodes represent criteria, and SRC stands for SOURCE.

It is always valid for inner join and cross joins to push (single source) criteria that are above the join, below the join. This allows for criteria originating in the user query to eventually be present in source queries below the joins. This result can be represented visually as:

```
JOIN - Inner Join on (A.x = B.x)
      /    \
     /   SELECT (B.y = 3)
    |        |
  SRC (A)   SRC (B)
```

The same optimization is valid for criteria specified against the outer side of an outer join. For example:

```
SELECT (B.y = 3)
       |
      JOIN - Right Outer Join on (A.x = B.x)
     /    \
  SRC (A)   SRC (B)
```

Becomes

```
      JOIN - Right Outer Join on (A.x = B.x)
      /    \
     /   SELECT (B.y = 3)
    |        |
  SRC (A)   SRC (B)
```

However criteria specified against the inner side of an outer join needs special consideration. The above scenario with a left or full outer join is not the same. For example:

```
SELECT (B.y = 3)
       |
      JOIN - Left Outer Join on (A.x = B.x)
```

```
    /    \
  SRC (A)   SRC (B)
```

Can become (available only after 5.0.2):

```
JOIN - Inner Join on (A.x = B.x)
      /   \
    /   SELECT (B.y = 3)
   |        |
  SRC (A)   SRC (B)
```

Since the criterion is not dependent upon the null values that may be populated from the inner side of the join, the criterion is eligible to be pushed below the join – but only if the join type is also changed to an inner join. On the other hand, criteria that are dependent upon the presence of null values CANNOT be moved. For example:

```
SELECT (B.y is null)
      |
     JOIN - Left Outer Join on (A.x = B.x)
    /    \
  SRC (A)   SRC (B)
```

The preceding plan tree must have the criteria remain above the join, becuase the outer join may be introducing null values itself.

- Raise Access – This rule attempts to raise the Access nodes as far up the plan as posssible. This is mostly done by looking at the source's capabilities and determining whether the operations can be achieved in the source or not.
- Raise Null – Raises null nodes. Raising a null node removes the need to consider any part of the old plan that was below the null node.
- Remove Optional Joins – Removes joins that are marked as or determined to be optional.
- Substitute Expressions – Used only when a function based index is present.
- Validate Where All – Ensures criteria is used when required by the source.

Cost calculations

The cost of node operations is primarily determined by an estimate of the number of rows (also referred to as cardinality) that will be processed by it. The optimizer will typically compute cardinalities from the bottom up of the plan (or subplan) at several points in time with planning – once generally with rule calculate cost, and then specifically for join planning and other decisions. The cost calculation is mainly directed by the statistics set on physical tables (cardinality, NNV, NDV, and so forth) and is also influenced by the presence of constraints (unique, primary key, index, and so forth). If there is a situation that seems like a sub-optimal plan is being chosen, you should first ensure that at least representative table cardinalities are set on the physical tables involved.

Reading a debug plan

As each relational sub plan is optimized, the plan will show what is being optimized and it's canonical form:

```
OPTIMIZE:
SELECT e1 FROM (SELECT e1 FROM pm1.g1) AS x
```

```
----------------------------------------------------------------------------
GENERATE CANONICAL:
SELECT e1 FROM (SELECT e1 FROM pm1.g1) AS x

CANONICAL PLAN:
Project(groups=[x], props={PROJECT_COLS=[e1]})
  Source(groups=[x], props={NESTED_COMMAND=SELECT e1 FROM pm1.g1, SYMBOL_MAP=
{x.e1=e1}})
    Project(groups=[pm1.g1], props={PROJECT_COLS=[e1]})
      Source(groups=[pm1.g1])
```

With more complicated user queries, such as a procedure invocation or one containing subqueries, the sub-plans may be nested within the overall plan. Each plan ends by showing the final processing plan:

```
----------------------------------------------------------------------------
OPTIMIZATION COMPLETE:
PROCESSOR PLAN:
AccessNode(0) output=[e1] SELECT g_0.e1 FROM pm1.g1 AS g_0
```

The affect of rules can be seen by the state of the plan tree before and after the rule fires. For example, the debug log below shows the application of rule merge virtual, which will remove the "x" inline view layer:

```
EXECUTING AssignOutputElements

AFTER:
Project(groups=[x], props={PROJECT_COLS=[e1], OUTPUT_COLS=[e1]})
  Source(groups=[x], props={NESTED_COMMAND=SELECT e1 FROM pm1.g1, SYMBOL_MAP=
{x.e1=e1}, OUTPUT_COLS=[e1]})
    Project(groups=[pm1.g1], props={PROJECT_COLS=[e1], OUTPUT_COLS=[e1]})
      Access(groups=[pm1.g1], props={SOURCE_HINT=null, MODEL_ID=Schema name=pm1,
nameInSource=null, uuid=3335, OUTPUT_COLS=[e1]})
        Source(groups=[pm1.g1], props={OUTPUT_COLS=[e1]})


============================================================================
EXECUTING MergeVirtual

AFTER:
Project(groups=[pm1.g1], props={PROJECT_COLS=[e1], OUTPUT_COLS=[e1]})
  Access(groups=[pm1.g1], props={SOURCE_HINT=null, MODEL_ID=Schema name=pm1,
nameInSource=null, uuid=3335, OUTPUT_COLS=[e1]})
    Source(groups=[pm1.g1])
```

Some important planning decisions are shown in the plan as they occur as an annotation. For example, the following code snippet shows that the access node could not be raised, because the parent **SELECT** node contained an unsupported subquery.

```
Project(groups=[pm1.g1], props={PROJECT_COLS=[e1], OUTPUT_COLS=null})
  Select(groups=[pm1.g1], props={SELECT_CRITERIA=e1 IN /*+ NO_UNNEST */(SELECT e1
FROM pm2.g1), OUTPUT_COLS=null})
    Access(groups=[pm1.g1], props={SOURCE_HINT=null, MODEL_ID=Schema name=pm1,
nameInSource=null, uuid=3341, OUTPUT_COLS=null})
      Source(groups=[pm1.g1], props={OUTPUT_COLS=null})
```

```
============================================================================
EXECUTING RaiseAccess
LOW Relational Planner SubqueryIn is not supported by source pm1 - e1 IN /*+ NO_UNNEST */
(SELECT e1 FROM pm2.g1) was not pushed

AFTER:
Project(groups=[pm1.g1])
  Select(groups=[pm1.g1], props={SELECT_CRITERIA=e1 IN /*+ NO_UNNEST */ (SELECT e1
FROM pm2.g1), OUTPUT_COLS=null})
    Access(groups=[pm1.g1], props={SOURCE_HINT=null, MODEL_ID=Schema name=pm1,
nameInSource=null, uuid=3341, OUTPUT_COLS=null})
      Source(groups=[pm1.g1])
```

Procedure planner

The procedure planner is fairly simple. It converts the statements in the procedure into instructions in a program that will be run during processing. This is mostly a 1-to-1 mapping and very little optimization is performed.

XQuery

XQuery is eligible for specific optimizations. For more information, see XQuery optimization. Document projection is the most common optimization. It will be shown in the debug plan as an annotation. For example, with the user query that contains "xmltable('/a/b' passing doc columns x string path '@x', val string path '.')", the debug plan would show a tree of the document that will effectively be used by the context and path XQuerys:

```
MEDIUM XQuery Planning Projection conditions met for /a/b - Document projection will be used
child element(Q{}a)
  child element(Q{}b)
    attribute attribute(Q{}x)
      child text()
    child text()
```

10.3. QUERY PLANS

When integrating information using a federated query planner it is useful to view the query plans to better understand how information is being accessed and processed, and to troubleshoot problems.

A query plan (also known as an execution or processing plan) is a set of instructions created by a query engine for executing a command submitted by a user or application. The purpose of the query plan is to execute the user's query in as efficient a way as possible.

Getting a query plan

You can get a query plan any time you execute a command. The following SQL options are available:

SET SHOWPLAN [ON|DEBUG]- Returns the processing plan or the plan and the full planner debug log. For more information, see *Reading a debug plan* in Query planner and *SET statement* in the Client Developer's Guide. With the above options, the query plan is available from the Statement object by casting to the **org.teiid.jdbc.TeiidStatement** interface or by using the SHOW PLAN statement. For more information, see *SHOW Statement* in the Client Developer's Guide. Alternatively you may use the EXPLAIN statement. For more information, see, Explain statement.

Retrieving a query plan using Data Virtualization extensions

```
statement.execute("set showplan on");
ResultSet rs = statement.executeQuery("select ...");
Data VirtualizationStatement tstatement = statement.unwrap(TeiidStatement.class);
PlanNode queryPlan = tstatement.getPlanDescription();
System.out.println(queryPlan);
```

Retrieving a query plan using statements

```
statement.execute("set showplan on");
ResultSet rs = statement.executeQuery("select ...");
...
ResultSet planRs = statement.executeQuery("show plan");
planRs.next();
System.out.println(planRs.getString("PLAN_XML"));
```

Retrieving a query plan using explain

```
ResultSet rs = statement.executeQuery("explain select ...");
System.out.println(rs.getString("QUERY PLAN"));
```

The query plan is made available automatically in several of Data Virtualization's tools.

Analyzing a query plan

After you obtain a query plan, you can examine it for the following items:

- Source pushdown – What parts of the query that got pushed to each source
  - Ensure that any predicates especially against indexes are pushed
- Joins – As federated joins can be quite expensive
  - Join ordering – Typically influenced by costing
  - Join criteria type mismatches.
  - Join algorithm used – Merge, enhanced merge, nested loop, and so forth.
- Presence of federated optimizations, such as dependent joins.
- Ensure hints have the desired affects. For more information about using hints, see the following additional resources:
  - *Hints and Options* in the Caching Guide.
  - *FROM clause hints* in FROM clause.
  - Subquery optimization.
  - Federated optimizations.

You can determine all of information in the preceding list from the processing plan. You will typically be interested in analyzing the textual form of the final processing plan. To understand why particular decisions are made for debugging or support you will want to obtain the full debug log which will contain

the intermediate planning steps as well as annotations as to why specific pushdown decisions are made.

A query plan consists of a set of nodes organized in a tree structure. If you are executing a procedure, the overall query plan will contain additional information related the surrounding procedural execution.

In a procedural context the ordering of child nodes implies the order of execution. In most other situation, child nodes may be executed in any order even in parallel. Only in specific optimizations, such as dependent join, will the children of a join execute serially.

Relational query plans

Relational plans represent the processing plan that is composed of nodes representing building blocks of logical relational operations. Relational processing plans differ from logical debug relational plans in that they will contain additional operations and execution specifics that were chosen by the optimizer.

The nodes for a relational query plan are:

- Access – Access a source. A source query is sent to the connection factory associated with the source. (For a dependent join, this node is called Dependent Access.)
- Dependent procedure access – Access a stored procedure on a source using multiple sets of input values.
- Batched update – Processes a set of updates as a batch.
- Project – Defines the columns returned from the node. This does not alter the number of records returned.
- Project into – Like a normal project, but outputs rows into a target table.
- Insert plan execution – Similar to a project into, but executes a plan rather than a source query. Typically created when executing an insert into view with a query expression.
- Window function project – Like a normal project, but includes window functions.
- Select – Select is a criteria evaluation filter node (WHERE / HAVING).
- Join – Defines the join type, join criteria, and join strategy (merge or nested loop).
- Union all – There are no properties for this node, it just passes rows through from it's children. Depending upon other factors, such as if there is a transaction or the source query concurrency allowed, not all of the union children will execute in parallel.
- Sort – Defines the columns to sort on, the sort direction for each column, and whether to remove duplicates or not.
- Dup remove – Removes duplicate rows. The processing uses a tree structure to detect duplicates so that results will effectively stream at the cost of IO operations.
- Grouping – Groups sets of rows into groups and evaluates aggregate functions.
- Null – A node that produces no rows. Usually replaces a Select node where the criteria is always false (and whatever tree is underneath). There are no properties for this node.
- Plan execution – Executes another sub plan. Typically the sub plan will be a non-relational plan.
- Dependent procedure execution – Executes a sub plan using multiple sets of input values.

- Limit – Returns a specified number of rows, then stops processing. Also processes an offset if present.
- XML table – Evaluates XMLTABLE. The debug plan will contain more information about the XQuery/XPath with regards to their optimization. For more information, see XQuery optimization.
- Text table - Evaluates TEXTTABLE
- Array table - Evaluates ARRAYTABLE
- Object table - Evaluates OBJECTTABLE

Node statistics

Every node has a set of statistics that are output. These can be used to determine the amount of data flowing through the node. Before execution a processor plan will not contain node statistics. Also the statistics are updated as the plan is processed, so typically you'll want the final statistics after all rows have been processed by the client.

Statistic	Description	Units
Node output rows	Number of records output from the node.	count
Node next batch process time	Time processing in this node only.	millisec
Node cumulative next batch process time	Time processing in this node + child nodes.	millisec
Node cumulative process time	Elapsed time from beginning of processing to end.	millisec
Node next batch calls	Number of times a node was called for processing.	count
Node blocks	Number of times a blocked exception was thrown by this node or a child.	count

In addition to node statistics, some nodes display cost estimates computed at the node.

Cost Estimates	Description	Units
Estimated node cardinality	Estimated number of records that will be output from the node; -1 if unknown.	count

The root node will display additional information.

Top level statistics	Description	Units
Data Bytes Sent	The size of the serialized data result (row and lob values) sent to the client.	bytes

Reading a processor plan

The query processor plan can be obtained in a plain text or XML format. The plan text format is typically easier to read, while the XML format is easier to process by tooling. When possible tooling should be used to examine the plans as the tree structures can be deeply nested.

Data flows from the leafs of the tree to the root. Sub plans for procedure execution can be shown inline, and are differentiated by different indentation. Given a user query of **SELECT pm1.g1.e1, pm1.g2.e2, pm1.g3.e3 from pm1.g1 inner join (pm1.g2 left outer join pm1.g3 on pm1.g2.e1=pm1.g3.e1) on pm1.g1.e1=pm1.g3.e1**, the text for a processor plan that does not push down the joins would look like:

```
ProjectNode
  + Output Columns:
    0: e1 (string)
    1: e2 (integer)
    2: e3 (boolean)
  + Cost Estimates:Estimated Node Cardinality: -1.0
  + Child 0:
    JoinNode
      + Output Columns:
        0: e1 (string)
        1: e2 (integer)
        2: e3 (boolean)
      + Cost Estimates:Estimated Node Cardinality: -1.0
      + Child 0:
        JoinNode
          + Output Columns:
            0: e1 (string)
            1: e1 (string)
            2: e3 (boolean)
          + Cost Estimates:Estimated Node Cardinality: -1.0
          + Child 0:
            AccessNode
              + Output Columns:e1 (string)
              + Cost Estimates:Estimated Node Cardinality: -1.0
              + Query:SELECT g_0.e1 AS c_0 FROM pm1.g1 AS g_0 ORDER BY c_0
              + Model Name:pm1
          + Child 1:
            AccessNode
              + Output Columns:
                0: e1 (string)
                1: e3 (boolean)
              + Cost Estimates:Estimated Node Cardinality: -1.0
              + Query:SELECT g_0.e1 AS c_0, g_0.e3 AS c_1 FROM pm1.g3 AS g_0 ORDER BY c_0
              + Model Name:pm1
          + Join Strategy:MERGE JOIN (ALREADY_SORTED/ALREADY_SORTED)
          + Join Type:INNER JOIN
```

```
        + Join Criteria:pm1.g1.e1=pm1.g3.e1
      + Child 1:
        AccessNode
          + Output Columns:
            0: e1 (string)
            1: e2 (integer)
          + Cost Estimates:Estimated Node Cardinality: -1.0
          + Query:SELECT g_0.e1 AS c_0, g_0.e2 AS c_1 FROM pm1.g2 AS g_0 ORDER BY c_0
          + Model Name:pm1
      + Join Strategy:ENHANCED SORT JOIN (SORT/ALREADY_SORTED)
      + Join Type:INNER JOIN
      + Join Criteria:pm1.g3.e1=pm1.g2.e1
  + Select Columns:
    0: pm1.g1.e1
    1: pm1.g2.e2
    2: pm1.g3.e3
```

Note that the nested join node is using a merge join and expects the source queries from each side to produce the expected ordering for the join. The parent join is an enhanced sort join which can delay the decision to perform sorting based upon the incoming rows. Note that the outer join from the user query has been modified to an inner join since none of the null inner values can be present in the query result.

The preceding plan can also be represented in in XML format as in the following example:

```
<?xml version="1.0" encoding="UTF-8"?>
<node name="ProjectNode">
  <property name="Output Columns">
    <value>e1 (string)</value>
    <value>e2 (integer)</value>
    <value>e3 (boolean)</value>
  </property>
  <property name="Cost Estimates">
    <value>Estimated Node Cardinality: -1.0</value>
  </property>
  <property name="Child 0">
    <node name="JoinNode">
      <property name="Output Columns">
        <value>e1 (string)</value>
        <value>e2 (integer)</value>
        <value>e3 (boolean)</value>
      </property>
      <property name="Cost Estimates">
        <value>Estimated Node Cardinality: -1.0</value>
      </property>
      <property name="Child 0">
        <node name="JoinNode">
          <property name="Output Columns">
            <value>e1 (string)</value>
            <value>e1 (string)</value>
            <value>e3 (boolean)</value>
          </property>
          <property name="Cost Estimates">
            <value>Estimated Node Cardinality: -1.0</value>
          </property>
          <property name="Child 0">
            <node name="AccessNode">
```

```
<property name="Output Columns">
  <value>e1 (string)</value>
</property>
<property name="Cost Estimates">
  <value>Estimated Node Cardinality: -1.0</value>
</property>
<property name="Query">
  <value>SELECT g_0.e1 AS c_0 FROM pm1.g1 AS g_0 ORDER BY c_0</value>
</property>
<property name="Model Name">
  <value>pm1</value>
</property>
</node>
</property>
<property name="Child 1">
<node name="AccessNode">
<property name="Output Columns">
  <value>e1 (string)</value>
  <value>e3 (boolean)</value>
</property>
<property name="Cost Estimates">
  <value>Estimated Node Cardinality: -1.0</value>
</property>
<property name="Query">
  <value>SELECT g_0.e1 AS c_0, g_0.e3 AS c_1 FROM pm1.g3 AS g_0
    ORDER BY c_0</value>
</property>
<property name="Model Name">
  <value>pm1</value>
</property>
</node>
</property>
<property name="Join Strategy">
  <value>MERGE JOIN (ALREADY_SORTED/ALREADY_SORTED)</value>
</property>
<property name="Join Type">
  <value>INNER JOIN</value>
</property>
<property name="Join Criteria">
  <value>pm1.g1.e1=pm1.g3.e1</value>
</property>
</node>
</property>
<property name="Child 1">
<node name="AccessNode">
<property name="Output Columns">
  <value>e1 (string)</value>
  <value>e2 (integer)</value>
</property>
<property name="Cost Estimates">
  <value>Estimated Node Cardinality: -1.0</value>
</property>
<property name="Query">
  <value>SELECT g_0.e1 AS c_0, g_0.e2 AS c_1 FROM pm1.g2 AS g_0
    ORDER BY c_0</value>
</property>
```

```
                <property name="Model Name">
                    <value>pm1</value>
                </property>
            </node>
        </property>
        <property name="Join Strategy">
            <value>ENHANCED SORT JOIN (SORT/ALREADY_SORTED)</value>
        </property>
        <property name="Join Type">
            <value>INNER JOIN</value>
        </property>
        <property name="Join Criteria">
            <value>pm1.g3.e1=pm1.g2.e1</value>
        </property>
    </node>
  </property>
  <property name="Select Columns">
    <value>pm1.g1.e1</value>
    <value>pm1.g2.e2</value>
    <value>pm1.g3.e3</value>
  </property>
</node>
```

Note that the same information appears in each of the plan forms. In some cases it can actually be easier to follow the simplified format of the debug plan final processor plan. The following output shows how the debug log represents the plan in the preceding XML example:

```
OPTIMIZATION COMPLETE:
PROCESSOR PLAN:
ProjectNode(0) output=[pm1.g1.e1, pm1.g2.e2, pm1.g3.e3] [pm1.g1.e1, pm1.g2.e2, pm1.g3.e3]
  JoinNode(1) [ENHANCED SORT JOIN (SORT/ALREADY_SORTED)] [INNER JOIN] criteria=
[pm1.g3.e1=pm1.g2.e1] output=[pm1.g1.e1, pm1.g2.e2, pm1.g3.e3]
    JoinNode(2) [MERGE JOIN (ALREADY_SORTED/ALREADY_SORTED)] [INNER JOIN] criteria=
[pm1.g1.e1=pm1.g3.e1] output=[pm1.g3.e1, pm1.g1.e1, pm1.g3.e3]
      AccessNode(3) output=[pm1.g1.e1] SELECT g_0.e1 AS c_0 FROM pm1.g1 AS g_0 ORDER BY
c_0
      AccessNode(4) output=[pm1.g3.e1, pm1.g3.e3] SELECT g_0.e1 AS c_0, g_0.e3 AS c_1 FROM
pm1.g3 AS g_0 ORDER BY c_0
    AccessNode(5) output=[pm1.g2.e1, pm1.g2.e2] SELECT g_0.e1 AS c_0, g_0.e2 AS c_1 FROM
pm1.g2 AS g_0 ORDER BY c_0
```

Common

- Output columns - what columns make up the tuples returned by this node.
- Data bytes sent - how many data byte, not including messaging overhead, were sent by this query.
- Planning time - the amount of time in milliseconds spent planning the query.

Relational

- Relational node ID – Matches the node ids seen in the debug log Node(id).
- Criteria – The Boolean expression used for filtering.

- Select columns – he columns that define the projection.
- Grouping columns – The columns used for grouping.
- Grouping mapping – Shows the mapping of aggregate and grouping column internal names to their expression form.
- Query – The source query.
- Model name – The model name.
- Sharing ID – Nodes sharing the same source results will have the same sharing id.
- Dependent join – If a dependent join is being used.
- Join strategy – The join strategy (Nested Loop, Sort Merge, Enhanced Sort, and so forth).
- Join type – The join type (Left Outer Join, Inner Join, Cross Join).
- Join criteria – The join predicates.
- Execution plan – The nested execution plan.
- Into target – The insertion target.
- Upsert – If the insert is an upsert.
- Sort columns – The columns for sorting.
- Sort mode – If the sort performs another function as well, such as distinct removal.
- Rollup – If the group by has the rollup option.
- Statistics – The processing statistics.
- Cost estimates – The cost/cardinality estimates including dependent join cost estimates.
- Row offset – The row offset expression.
- Row limit – The row limit expression.
- With – The with clause.
- Window functions – The window functions being computed.
- Table function – The table function (XMLTABLE, OBJECTTABLE, TEXTTABLE, and so forth).
- Streaming – If the XMLTABLE is using stream processing.

Procedure

- Expression
- Result Set
- Program
- Variable

- Then
- Else

Other plans

Procedure execution (including instead of triggers) use intermediate and final plan forms that include relational plans. Generally the structure of the XML/procedure plans will closely match their logical forms. It's the nested relational plans that will be of interest when analyzing performance issues.

10.4. FEDERATED OPTIMIZATIONS

Access patterns

Access patterns are used on both physical tables and views to specify the need for criteria against a set of columns. Failure to supply the criteria will result in a planning error, rather than a run-away source query. Access patterns can be applied in a set such that only one of the access patterns is required to be satisfied.

Currently any form of criteria referencing an affected column may satisfy an access pattern.

Pushdown

In federated database systems pushdown refers to decomposing the user level query into source queries that perform as much work as possible on their respective source system. Pushdown analysis requires knowledge of source system capabilities, which is provided to Data Virtualization though the Connector API. Any work not performed at the source is then processed in Federate's relational engine.

Based upon capabilities, Data Virtualization will manipulate the query plan to ensure that each source performs as much joining, filtering, grouping, and so forth, as possible. In many cases, such as with join ordering, planning is a combination of standard relational techniques along with costing information, and heuristics for pushdown optimization.

Criteria and join push down are typically the most important aspects of the query to push down when performance is a concern. For information about how to read a plan to ensure that source queries are as efficient as possible, see Query plans.

Dependent joins

A special optimization called a dependent join is used to reduce the rows returned from one of the two relations involved in a multi-source join. In a dependent join, queries are issued to each source sequentially rather than in parallel, with the results obtained from the first source used to restrict the records returned from the second. Dependent joins can perform some joins much faster by drastically reducing the amount of data retrieved from the second source and the number of join comparisons that must be performed.

The conditions when a dependent join is used are determined by the query planner based on access patterns, hints, and costing information. You can use the following types of dependent joins with Data Virtualization:

Join based on equality or inequality

Where the engine determines how to break up large queries based on translator capabilities.

Key pushdown

Where the translator has access to the full set of key values and determines what queries to send.

Full pushdown

Where the translator ships the all data from the independent side to the translator. Can be used automatically by costing or can be specified as an option in the hint.

You can use the following types of hints in Data Virtualization to control dependent join behavior:

MAKEIND

Indicates that the clause should be the independent side of a dependent join.

MAKEDEP

Indicates that the clause should be the dependent side of a join. As a non-comment hint, you can use **MAKEDEP** with the following optional **MAX** and **JOIN** arguments.

MAKEDEP(JOIN)

meaning that the entire join should be pushed.

MAKEDEP(MAX:5000)

meaning that the dependent join should only be performed if there are less than the maximum number of values from the independent side.

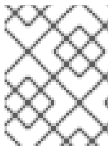
MAKENOTDEP

Prevents the clause from being the dependent side of a join.

These can be placed in either the OPTION Clause or directly in the FROM Clause. As long as all Access Patterns can be met, the MAKEIND, MAKEDEP, and MAKENOTDEP hints override any use of costing information. MAKENOTDEP supersedes the other hints.

TIP

The MAKEDEP or MAKEIND hints should only be used if the proper query plan is not chosen by default. You should ensure that your costing information is representative of the actual source cardinality.



NOTE

An inappropriate MAKEDEP or MAKEIND hint can force an inefficient join structure and may result in many source queries.

TIP

While these hints can be applied to views, the optimizer will by default remove views when possible. This can result in the hint placement being significantly different than the original intention. You should consider using the NO_UNNEST hint to prevent the optimizer from removing the view in these cases.

In the simplest scenario the engine will use IN clauses (or just equality predicates) to filter the values coming from the dependent side. If the number of values from the independent side exceeds the translators **MaxInCriteriaSize**, the values will be split into multiple IN predicates up to **MaxDependentPredicates**. When the number of independent values exceeds **MaxInCriteriaSize*MaxDependentPredicates**, then multiple dependent queries will be issued in parallel.

If the translator returns true for **supportsDependentJoins**, then the engine may provide the entire set of independent key values. This will occur when the number of independent values exceeds **MaxInCriteriaSize*MaxDependentPredicates** so that the translator may use specific logic to avoid issuing multiple queries as would happen in the simple scenario.

If the translator returns true for both **supportsDependentJoins** and **supportsFullDependentJoins**

then a full pushdown may be chosen by the optimizer A full pushdown, sometimes also called as data-ship pushdown, is where all the data from independent side of the join is sent to dependent side. This has an added benefit of allowing the plan above the join to be eligible for pushdown as well. This is why the optimizer may choose to perform a full pushdown even when the number of independent values does not exceed **MaxInCriteriaSize*MaxDependentPredicates**. You may also force full pushdown using the **MAKEDEP(JOIN)** hint. The translator is typically responsible for creating, populating, and removing a temporary table that represents the independent side. For more information about how to use custom translators with dependent, key, and full pushdown, see *Dependent join pushdown* in Translator Development. NOTE: By default, **Key/Full Pushdown** is compatible with only a subset of JDBC translators. To use it, set the translator override property **enableDependentJoins** to **true**. The JDBC source must allow for the creation of temporary tables, which typically requires a Hibernate dialect. The following translators are compatible with this feature: DB2, Derby, H2, Hana, HSQL, MySQL, Oracle, PostgreSQL, SQL Server, SAP IQ, Sybase, Teiid, and Teradata.

Copy criteria

Copy criteria is an optimization that creates additional predicates based upon combining join and where clause criteria. For example, equi-join predicates **(source1.table.column = source2.table.column)** are used to create new predicates by substituting **source1.table.column** for **source2.table.column** and vice versa. In a cross-source scenario, this allows for where criteria applied to a single side of the join to be applied to both source queries

Projection minimization

Data Virtualization ensures that each pushdown query only projects the symbols required for processing the user query. This is especially helpful when querying through large intermediate view layers.

Partial aggregate pushdown

Partial aggregate pushdown allows for grouping operations above multi-source joins and unions to be decomposed so that some of the grouping and aggregate functions may be pushed down to the sources.

Optional join

An optional or redundant join is one that can be removed by the optimizer. The optimizer will automatically remove inner joins based upon a foreign key or left outer joins when the outer results are unique.

The optional join hint goes beyond the automatic cases to indicate to the optimizer that a joined table should be omitted if none of its columns are used by the output of the user query or in a meaningful way to construct the results of the user query. This hint is typically only used in view layers containing multi-source joins.

The optional join hint is applied as a comment on a join clause. It can be applied in both ANSI and non-ANSI joins. With non-ANSI joins an entire joined table may be marked as optional.

Example: Optional join hint

```
select a.column1, b.column2 from a, /*+ optional */ b WHERE a.key = b.key
```

Suppose this example defines a view layer **X**. If **X** is queried in such a way as to not need **b.column2**, then the optional join hint will cause **b** to be omitted from the query plan. The result would be the same as if **X** were defined as:

Example: Optional join hint

```
select a.column1 from a
```

Example: ANSI optional join hint

```
select a.column1, b.column2, c.column3 from /*+ optional */ (a inner join b ON a.key = b.key) INNER
JOIN c ON a.key = c.key
```

In this example the ANSI join syntax allows for the join of a and b to be marked as optional. Suppose this example defines a view layer X. Only if both **column a.column1** and **b.column2** are not needed, for example, **SELECT column3 FROM X** will the join be removed.

The optional join hint will not remove a bridging table that is still required.

Example: Bridging table

```
select a.column1, b.column2, c.column3 from /*+ optional */ a, b, c WHERE ON a.key = b.key AND
a.key = c.key
```

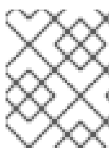
Suppose this example defines a view layer **X**. If **b.column2** or **c.column3** are solely required by a query to **X**, then the join on a be removed. However if **a.column1** or both **b.column2** and **c.column3** are needed, then the optional join hint will not take effect.

When a join clause is omitted via the optional join hint, the relevant criteria is not applied. Thus it is possible that the query results may not have the same cardinality or even the same row values as when the join is fully applied.

Left/right outer joins where the inner side values are not used and whose rows under go a distinct operation will automatically be treated as an optional join and do not require a hint.

Example: Unnecessary optional join hint

```
select distinct a.column1 from a LEFT OUTER JOIN /*+optional*/ b ON a.key = b.key
```



NOTE

A simple "SELECT COUNT(*) FROM VIEW" against a view where all join tables are marked as optional will not return a meaningful result.

Source hints

Data Virtualization user and transformation queries can contain a meta source hint that can provide additional information to source queries. The source hint has the following form:

```
/*+ sh[[ KEEP ALIASES]:'arg'] source-name[ KEEP ALIASES]:'arg1' ... */
```

- The source hint is expected to appear after the query (SELECT, INSERT, UPDATE, DELETE) keyword.
- Source hints can appear in any subquery, or in views. All hints applicable to a given source query will be collected and pushed down together as a list. The order of the hints is not guaranteed.
- The sh arg is optional and is passed to all source queries via the **ExecutionContext.getGeneralHints** method. The additional args should have a source-name

that matches the source name assigned to the translator in the VDB. If the source-name matches, the hint values will be supplied via the **ExecutionContext.getSourceHints** method. For more information about using an ExecutionContext, see Translator Development .

- Each of the arg values has the form of a string literal - it must be surrounded in single quotes and a single quote can be escaped with another single quote. Only the Oracle translator does anything with source hints by default. The Oracle translator will use both the source hint and the general hint (in that order) if available to form an Oracle hint enclosed in /*+ ... */.

- If the KEEP ALIASES option is used either for the general hint or on the applicable source specific hint, then the table/view aliases from the user query and any nested views will be preserved in the push-down query. This is useful in situations where the source hint may need to reference aliases and the user does not wish to rely on the generated aliases (which can be seen in the query plan in the relevant source queries – see above). However in some situations this may result in an invalid source query if the preserved alias names are not valid for the source or result in a name collision. If the usage of KEEP ALIASES results in an error, the query could be modified by preventing view removal with the NO_UNNEST hint, the aliases modified, or the KEEP ALIASES option could be removed and the query plan used to determine the generated alias names.

Sample source hints

```
SELECT /*+ sh:'general hint' */ ...

SELECT /*+ sh KEEP ALIASES:'general hint' my-oracle:'oracle hint' */ ...
```

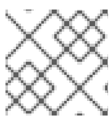
Partitioned union

Union partitioning is inferred from the transformation/inline view. If one (or more) of the UNION columns is defined by constants and/or has WHERE clause IN predicates containing only constants that make each branch mutually exclusive, then the UNION is considered partitioned. UNION ALL must be used and the UNION cannot have a LIMIT, WITH, or ORDER BY clause (although individual branches may use LIMIT, WITH, or ORDER BY). Partitioning values should not be null.

Example: Partitioned union

```
create view part as select 1 as x, y from foo union all select z, a from foo1 where z in (2, 3)
```

The view is partitioned on column x, since the first branch can only be the value 1 and the second branch can only be the values 2 or 3.



NOTE

More advanced or explicit partitioning will be considered for future releases.

The concept of a partitioned union is used for performing partition-wise joins, in Updatable Views, and Partial Aggregate Pushdown. These optimizations are also applied when using the multi-source feature as well - which introduces an explicit partitioning column.

Partition-wise joins take a join of unions and convert the plan into a union of joins, such that only matching partitions are joined against one another. If you want a partition-wise join to be performed implicit without the need for an explicit join predicate on the partitioning column, set the model/schema property **implicit_partition.columnName** to name of the partitioning column used on each partitioned view in the model/schema.

```
CREATE VIRTUAL SCHEMA all_customers SERVER server OPTIONS
("implicit_partition.columnName" 'theColumn');
```

Standard relational techniques

Data Virtualization also incorporates many standard relational techniques to ensure efficient query plans.

- Rewrite analysis for function simplification and evaluation.
- Boolean optimizations for basic criteria simplification.
- Removal of unnecessary view layers.
- Removal of unnecessary sort operations.
- Advanced search techniques through the left-linear space of join trees.
- Parallelizing of source access during execution.
- Subquery optimization.

Join compensation

Some source systems only allow "relationship" queries logically producing left outer join results. Even when queried with an inner join, Data Virtualization will attempt to form an appropriate left outer join. These sources are restricted to use with key joins. In some circumstances foreign key relationships on the same source should not be traversed at all or with the referenced table on the outer side of join. The extension property **teiid_rel:allow-join** can be used on the foreign key to further restrict the pushdown behavior. With a value of "false" no join pushdown will be allowed, and with a value of "inner" the referenced table must be on the inner side of the join. If the join pushdown is prevented, the join will be processed as a federated join.

10.5. SUBQUERY OPTIMIZATION

- EXISTS subqueries are typically rewrite to "SELECT 1 FROM ..." to prevent unnecessary evaluation of SELECT expressions.
- Quantified compare SOME subqueries are always turned into an equivalent IN predicate or comparison against an aggregate value. e.g. col > SOME (select col1 from table) would become col > (select min(col1) from table)
- Uncorrelated EXISTs and scalar subquery that are not pushed to the source can be pre-evaluated prior to source command formation.
- Correlated subqueries used in DETELEs or UPDATEs that are not pushed as part of the corresponding DELETE/UPDATE will cause Data Virtualization to perform row-by-row compensating processing.
- The merge join (MJ) hint directs the optimizer to use a traditional, semijoin, or antisemijoin merge join if possible. The dependent join (DJ) is the same as the MJ hint, but additionally directs the optimizer to use the subquery as the independent side of a dependent join if possible. This will only happen if the affected table has a primary key. If it does not, then an exception will be thrown.
- WHERE or HAVING clause IN, Quantified Comparison, Scalar Subquery Compare, and EXISTs predicates can take the MJ, DJ, or NO_UNNEST (no unnest) hints appearing just before the

subquery. The NO_UNNEST hint, which supersedes the other hints, will direct the optimizer to leave the subquery in place.

- SELECT scalar subqueries can take the MJ or NO_UNNEST hints appearing just before the subquery. The MJ hint directs the optimizer to use a traditional or semijoin merge join if possible. The NO_UNNEST hint, which supersedes the other hints, will direct the optimizer to leave the subquery in place.

Merge join hint usage

```
SELECT col1 from tbl where col2 IN /*+ MJ*/ (SELECT col1 FROM tbl2)
```

Dependent join hint usage

```
SELECT col1 from tbl where col2 IN /*+ DJ */ (SELECT col1 FROM tbl2)
```

No unnest hint usage

```
SELECT col1 from tbl where col2 IN /*+ NO_UNNEST */ (SELECT col1 FROM tbl2)
```

- The system property **org.teiid.subqueryUnnestDefault** controls whether the optimizer will by default unnest subqueries during rewrite. If **true**, then most non-negated WHERE or HAVING clause EXISTS or IN subquery predicates can be converted to a traditional join.
- The planner will always convert to antijoin or semijoin variants if costing is favorable. Use a hint to override this behavior needed.
- EXISTs and scalar subqueries that are not pushed down, and not converted to merge joins, are implicitly limited to 1 and 2 result rows respectively via a limit.
- Conversion of subquery predicates to nested loop joins is not yet available.

10.6. XQUERY OPTIMIZATION

A technique known as document projection is used to reduce the memory footprint of the context item document. Document projection loads only the parts of the document needed by the relevant XQuery and path expressions. Since document projection analysis uses all relevant path expressions, even 1 expression that could potentially use many nodes, for example, **//x rather than /a/b/x** will cause a larger memory footprint. With the relevant content removed the entire document will still be loaded into memory for processing. Document projection will only be used when there is a context item (unnamed PASSING clause item) passed to XMLTABLE/XMLQUERY. A named variable will not have document projection performed. In some cases the expressions used may be too complex for the optimizer to use document projection. You should check the SHOWPLAN DEBUG full plan output to see if the appropriate optimization has been performed.

With additional restrictions, simple context path expressions allow the processor to evaluate document subtrees independently - without loading the full document in memory. A simple context path expression can be of the form **[/][ns:]root/[ns1:]elem/…**`, where a namespace prefix or element name can also be the * wild card. As with normal XQuery processing if namespace prefixes are used in the XQuery expression, they should be declared using the XMLNAMESPACES clause.

Streaming eligible XMLQUERY

```
XMLQUERY('/*:root/*:child' PASSING doc)
```

–

Rather than loading the entire doc in-memory as a DOM tree, each child element will be independently added to the result.

Streaming ineligible XMLQUERY

```
XMLQUERY('//child' PASSING doc)
```

The use of the descendant axis prevents the streaming optimization, but document projection can still be performed.

When using XMLTABLE, the COLUMN PATH expressions have additional restrictions. They are allowed to reference any part of the element subtree formed by the context expression and they may use any attribute value from their direct parentage. Any path expression where it is possible to reference a non-direct ancestor or sibling of the current context item prevent streaming from being used.

Streaming eligible XMLTABLE

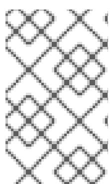
```
XMLTABLE('/*:root/*:child' PASSING doc COLUMNS fullchild XML PATH '.', parent_attr string PATH
'../@attr', child_val integer)
```

The context XQuery and the column path expression allow the streaming optimization, rather than loading the entire document in-memory as a DOM tree, each child element will be independently added to the result.

Streaming ineligible XMLTABLE

```
XMLTABLE('/*:root/*:child' PASSING doc COLUMNS sibling_attr string PATH '../other_child/@attr')
```

The reference of an element outside of the child subtree in the sibling_attr path prevents the streaming optimization from being used, but document projection can still be performed.



NOTE

Column paths should be as targeted as possible to avoid performance issues. A general path such as **..//child** will cause the entire subtree of the context item to be searched on each output row.

10.7. FEDERATED FAILURE MODES

Data Virtualization provides the capability to obtain *partial results* in the event of data source unavailability or failure. This is especially useful when unioning information from multiple sources, or when doing a left outer join, where you are appending columns to a master record, but still want the record if the extra information is not available.

A source is considered to be unavailable if the connection factory that is associated with the source generates an exception in response to a query. The exception will be propagated to the query processor, where it will become a warning on the statement. For more information about partial results and SQLWarning objects, see *Partial Results Mode* in the Client Developer's Guide.

10.8. CONFORMED TABLES

A conformed table is a source table that is the same in several physical sources.

Typically this would be used when reference data exists in multiple sources, but only a single metadata entry is desired to represent the table.

Conformed tables are defined by adding the following extension metadata property to the appropriate source tables:

```
{http://www.teiid.org/ext/relational/2012}conformed-sources
```

You can set extension properties in the DDL file by using full DDL metadata or alter statements, or at runtime by using the **setProperty** system procedure. The property is expected to be a comma separated list of physical model/schema names.

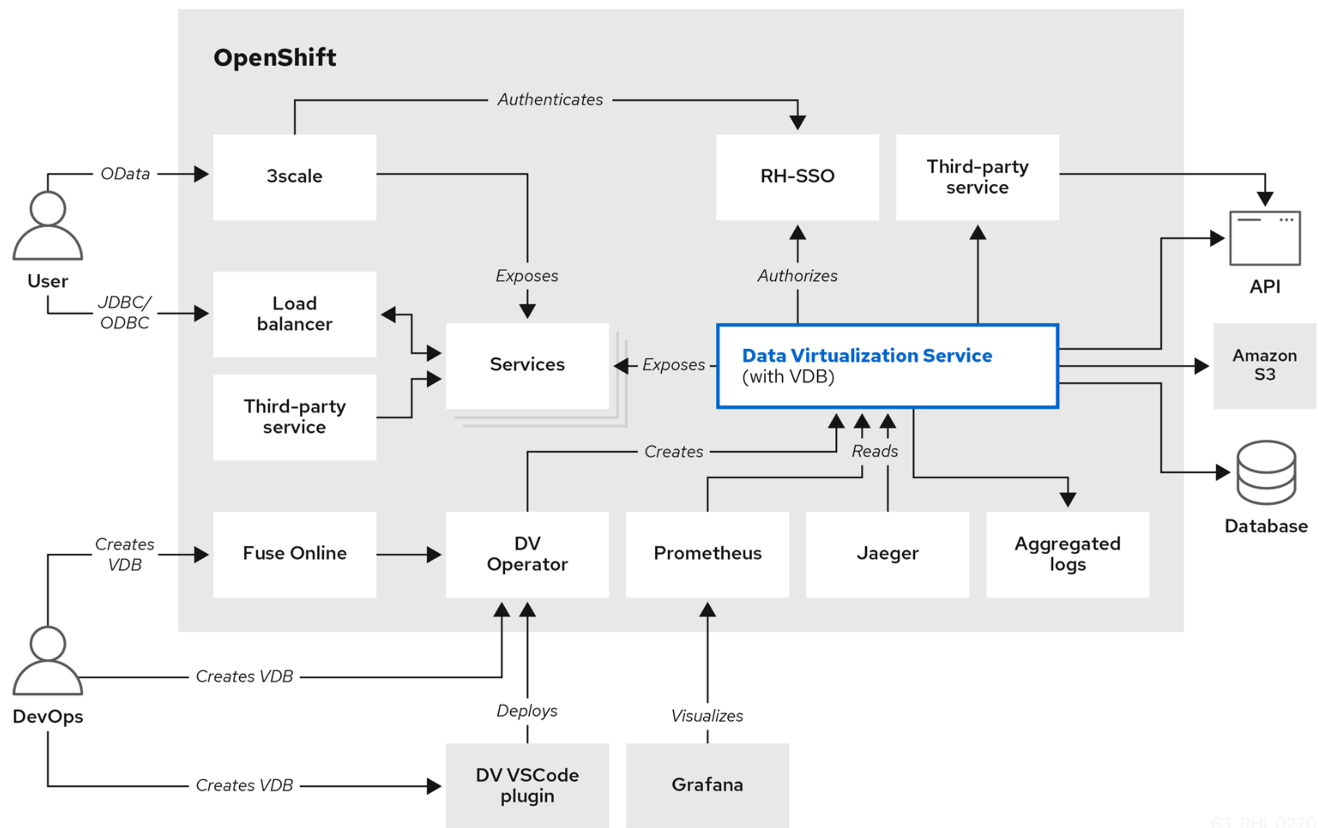
DDL alter example

```
ALTER FOREIGN TABLE "reference_data" OPTIONS (ADD "teiid_rel:conformed-sources"
'source2,source3');
```

There is no expectation that a metadata entry exists on the other schemas.

The engine will take the list of conformed sources and associate a set of model metadata ids to the corresponding access node. The logic considering joins and subqueries will also consider the conformed sets when making pushdown decisions. The subquery handling will only check for conformed sources for the subquery – not in the parent. So having a conformed table in the subquery will pushdown as expected, but not vice versa.

CHAPTER 11. DATA VIRTUALIZATION ARCHITECTURE



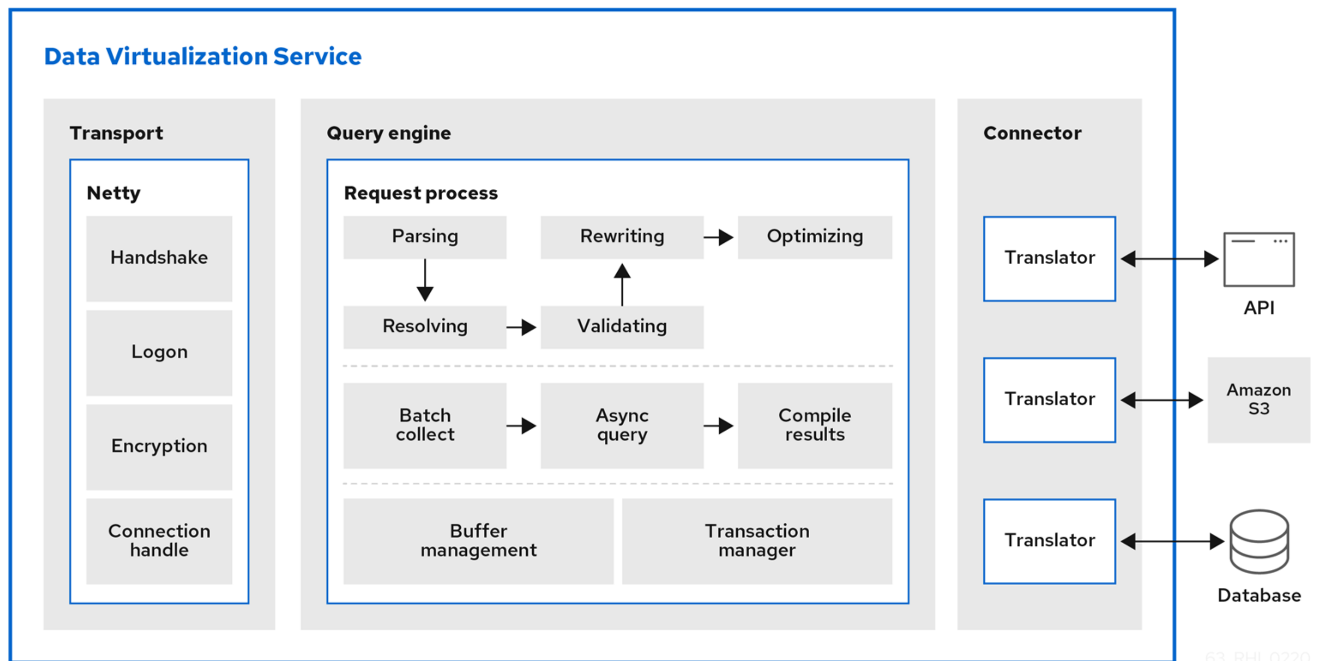
Transport

Transport services manage client connections: security authentication, encryption, and so forth.

Query Engine

The query engine has several layers and components. At a high level, request processing is structured as follows:

The following diagram shows the components that make up the data virtualization service in greater detail:



1. SQL is converted to a processor plan. The engine receives an incoming SQL query. It is parsed to a internal command. Then the command is converted a logical plan via resolving, validating, and rewriting. Finally, rule and cost-based optimization convert the logical plan to a final processor plan. For more information, see Federated planning.

2. Batch processing. The source and other aspects of query processing may return results asynchronously to the processing thread. As soon as possible, batches of results are made available to the client.

3. Buffer management controls the bulk of the on and off heap memory that Data Virtualization is using. It prevents consuming too much memory that otherwise might exceed the VM size.

4. Transaction management determines when transactions are needed and interacts with the TransactionManager subsystem to coordinate XA transactions.
   Source queries are handled by the data tier layer which interfaces with the query engine and the connector layer which utilizes a translator to interact directly with a source. Connectivity is provided for heterogeneous data stores, such as databases or data warehouses, NoSQL, Hadoop, data grid/cache, files, SaaS, and so on.

   Translator

   Data Virtualization has developed a series of translators. For more information, see Translators.

11.1. TERMINOLOGY

VM or process

A Spring Boot instance of Data Virtualization.

Host

A machine that runs one or more VMs.

Service

A subsystem that runs in a VM (often in many VMs) and provides a related set of functionality. In addition to these main components, the service platform makes the following core set of services available to the applications that are built on top of it:

Session

The Session service manages active session information.

Buffer manager

The Buffer Manager service provides access to data management for intermediate results. For more information, see *Buffer management* in Data management.

Transaction

The Transaction service manages global, local, and request scoped transactions. For more information, see Transactions.

11.2. DATA MANAGEMENT

Cursoring and batching

Data Virtualization cursors all results, regardless of whether they are from one source or many sources, and regardless of what type of processing (joins, unions, and so forth.) have been performed on the results.

Data Virtualization processes results in batches. A batch is simply a set of records. The number of rows in a batch is determined by the buffer system property *processor-batch-size* and is scaled upon the estimated memory footprint of the batch.

Client applications have no direct knowledge of batches or batch sizes, but rather specify fetch size. However the first batch, regardless of fetch size is always proactively returned to synchronous clients. Subsequent batches are returned based on client demand for the data. Pre-fetching is utilized at both the client and connector levels.

Buffer management

The buffer manager manages memory for all result sets used in the query engine. That includes result sets read from a connection factory, result sets used temporarily during processing, and result sets prepared for a user. Each result set is referred to in the buffer manager as a tuple source.

When retrieving batches from the buffer manager, the size of a batch in bytes is estimated and then allocated against the max limit.

Memory management

The buffer manager has two storage managers - a memory manager and a disk manager. The buffer manager maintains the state of all the batches, and determines when batches must be moved from memory to disk.

Disk management

Each tuple source has a dedicated file (named by the ID) on disk. This file will be created only if at least one batch for the tuple source had to be swapped to disk. The file is random access. The processor batch size property defines how many rows should nominally exist in a batch assuming 2048 bits worth of data in a row. If the row is larger or smaller than that target, the engine will adjust the batch size for those tuples accordingly. Batches are always read and written from the storage manager whole.

The disk storage manager caps the maximum number of open files to prevent running out of file handles. In cases with heavy buffering, this can cause wait times while waiting for a file handle to become available (the default maximum open files is 64).

Cleanup

When a tuple source is no longer needed, it is removed from the buffer manager. The buffer manager will remove it from both the memory storage manager and the disk storage manager. The disk storage manager will delete the file. In addition, every tuple source is tagged with a "group name" which is typically the session ID of the client. When the client's session is terminated (by closing the connection, server detecting client shutdown, or administrative termination), a call is sent to the buffer manager to remove all tuple sources for the session.

In addition, when the query engine is shutdown, the buffer manager is shut down, which will remove all state from the disk storage manager and cause all files to be closed. When the query engine is stopped, it is safe to delete any files in the buffer directory as they are not used across query engine restarts and must be due to a system crash where buffer files were not cleaned up.

11.3. QUERY TERMINATION

Canceling queries

When a query is canceled, processing will be stopped in the query engine and in all connectors involved in the query. The semantics of what a connector does in response to a cancellation command depends on the connector implementation. For example, JDBC connectors will asynchronously call **cancel** on the underlying JDBC driver, which might or might not be compatible with this method.

User query timeouts

User query timeouts in Data Virtualization can be managed on the client-side or the server-side. Timeouts are only relevant for the first record returned. If the first record has not been received by the client within the specified timeout period, a **cancel** command is issued to the server for the request and no results are returned to the client. The cancel command is issued asynchronously without the client's intervention.

The JDBC API uses the query timeout set by the **java.sql.Statement.setQueryTimeout** method. You can also set a default statement timeout via the connection property "QUERYTIMEOUT". ODBC clients can also use QUERYTIMEOUT as an execution property via a set statement to control the default timeout setting. See the Client Developer's Guide for more on connection/execution properties and set statements.

Server-side timeouts start when the query is received by the engine. Network latency or server load can delays the processing of I/O work after a client issues the query. The timeout will be cancelled if the first result is sent back before the timeout has ended. For more information about setting the **query-timeout** property for a virtual database, see Virtual database properties. For more information about modifying system properties to set a default query timeout for all queries, see *System properties* in the Administrator's Guide.

11.4. PROCESSING

Join algorithms

Nested loop does the most obvious processing. For every row in the outer source, it compares with every row in the inner source. Nested loop is only used when the join criteria has no equi-join predicates.

A merge join first sorts the input sources on the joined columns. You can then walk through each side in parallel (effectively one pass through each sorted source), and when you have a match, emit a row. In

general, merge join is on the order of n+m rather than n*m in nested loop. Merge join is the default algorithm.

Using costing information the engine may also delay the decision to perform a full sort merge join. Based upon the actual row counts involved, the engine can choose to build an index of the smaller side (which will perform similarly to a hash join) or to only partially sort the larger side of the relation.

Joins involving equi-join predicates can be converted into dependent joins. For more information, see *Dependent joins* in Federated optimizations.

Sort-based algorithms

Sorting is used as the basis of the Sort (ORDER BY), Grouping (GROUP BY), and DupRemoval (SELECT DISTINCT) operations. The sort algorithm is a multi-pass merge-sort that does not require all of the result set to ever be in memory, yet uses the maximal amount of memory allowed by the buffer manager.

Sorting consists of two phases. In the first phase ("sort"), the algorithm processes an unsorted input stream to produce one or more sorted input streams. Each pass reads as much of the unsorted stream as possible, sorts it, and writes it back out as a new stream. When an unsorted stream is processed, the resulting sorted stream might be too large to reside in memory. If the size of a sorted stream exceeds the available memory, it is written out to multiple sorted streams.

The second phase of the sort algorithm ("merge") consists of a set of phases that grab the next batch from as many sorted input streams as will fit in memory. It then repeatedly grabs the next tuple in sorted order from each stream and outputs merged sorted batches to a new sorted stream. At completion of the phase, all input streams are dropped. In this way, each phase reduces the number of sorted streams. When only one stream remains, it is the final output.

CHAPTER 12. BNF FOR SQL GRAMMAR

- Main Entry Points
  - callable statement
  - ddl statement
  - explain
  - directly executable statement
- Reserved Keywords
- Non-Reserved Keywords
- Reserved Keywords For Future Use
- Tokens
- Production Cross-Reference
- Productions

12.1. RESERVED KEYWORDS

Keyword	Usage
ADD	add set child option, add set option, ADD column, ADD constraint
ALL	standard aggregate function, CREATE POLICY, function, GRANT, query expression body, query term, Revoke GRANT, select clause, quantified comparison predicate
ALTER	alter, ALTER PROCEDURE , alterStatement, ALTER TABLE , grant type
AND	between predicate, boolean term, window frame
ANY	standard aggregate function, with role, quantified comparison predicate
ARRAY	ARRAY expression constructor
ARRAY_AGG	ordered aggregate function

Keyword	Usage
AS	alter, ALTER PROCEDURE , ALTER TABLE , ALTER TRIGGER , array table, create procedure, create a domain or type alias, option namespace, create trigger, create view, delete statement, derived column, dynamic data statement, function, json table, loop statement, xml namespace element, object table, select derived column, table subquery, text table, table name, unescapedFunction, update statement, with list element, xml serialize, xml table
ASC	sort specification
ATOMIC	compound statement, for each row trigger action
AUTHENTICATED	with role
BEGIN	compound statement, for each row trigger action
BETWEEN	between predicate, window frame
BIGDECIMAL	simple data type
BIGINT	simple data type
BIGINTEGER	simple data type
BLOB	simple data type, xml serialize
BOOLEAN	simple data type
BOTH	function
BREAK	branching statement
BY	group by clause, order by clause, window specification
BYTE	simple data type
CALL	callable statement, call statement
CASE	case expression, searched case expression
CAST	function
CHAR	function, simple data type

Keyword	Usage
CLOB	simple data type, xml serialize
COLUMN	ADD column, DROP column, ALTER TABLE , GRANT, Revoke GRANT
COMMIT	create temporary table
CONSTRAINT	GRANT, table constraint
CONTINUE	branching statement
CONVERT	function
CREATE	create procedure, create data wrapper, create database, create a domain or type alias, create foreign temp table, CREATE POLICY, create role , create schema, create server, create table, create temporary table, create trigger
CROSS	cross join
CUME_DIST	analytic aggregate function
CURRENT_DATE	function
CURRENT_TIME	function
CURRENT_TIMESTAMP	function
DATE	non numeric literal, simple data type
DAY	function
DECIMAL	simple data type
DECLARE	declare statement
DELETE	alter, ALTER TRIGGER , CREATE POLICY, create trigger, delete statement, grant type
DESC	sort specification
DISTINCT	standard aggregate function, function, is distinct, query expression body, query term, select clause
DOUBLE	simple data type

Keyword	Usage
DROP	DROP column, drop option, Drop data wrapper, drop option, DROP POLICY, drop procedure, drop role , drop schema, drop server, drop table, drop table, grant type
EACH	for each row trigger action
ELSE	case expression, if statement, searched case expression
END	case expression, compound statement, for each row trigger action, searched case expression
ERROR	raise error statement
ESCAPE	match predicate, text table
EXCEPT	query expression body
EXEC	dynamic data statement, call statement
EXECUTE	dynamic data statement, grant type, call statement
EXISTS	exists predicate
FALSE	explain option, json table, non numeric literal
FETCH	fetch clause
FILTER	filter clause
FLOAT	simple data type
FOR	CREATE POLICY, for each row trigger action, function, json table column, text aggreate function, text table column, xml table column
FOREIGN	ALTER PROCEDURE , ALTER TABLE , create procedure, create data wrapper, create foreign or global temporary table, create foreign temp table, create schema, create server, Drop data wrapper, drop procedure, drop schema, drop table, foreign key, Import foreign schema, with role
FROM	delete statement, from clause, function, Import foreign schema, is distinct, Revoke GRANT

Keyword	Usage
FULL	qualified table
FUNCTION	create procedure, drop procedure, GRANT, Revoke GRANT
GLOBAL	create foreign or global temporary table, drop table
GRANT	GRANT
GROUP	function, group by clause
HANDLER	create data wrapper
HAVING	having clause
HOUR	function
IF	if statement
IMMEDIATE	dynamic data statement
IMPORT	Import another Database, Import foreign schema
IN	function, procedure parameter, in predicate
INNER	qualified table
INOUT	procedure parameter
INSERT	alter, ALTER TRIGGER, CREATE POLICY, create trigger, function, insert statement, grant type
INTEGER	simple data type
INTERSECT	query term
INTO	dynamic data statement, Import foreign schema, insert statement, into clause
IS	is distinct, is null predicate
JOIN	cross join, make dep options, qualified table
LANGUAGE	GRANT, object table, Revoke GRANT

Keyword	Usage
LATERAL	table subquery
LEADING	function
LEAVE	branching statement
LEFT	function, qualified table
LIKE	match predicate
LIKE_REGEX	like regex predicate
LIMIT	limit clause
LOCAL	create foreign temp table, create temporary table
LONG	simple data type
LOOP	loop statement
MAKEDEP	option clause, table primary
MAKEIND	option clause, table primary
MAKENOTDEP	option clause, table primary
MERGE	insert statement
MINUTE	function
MONTH	function
NO	make dep options, xml namespace element, text aggregate function, text table column, text table
NOCACHE	option clause
NOT	alter column options, between predicate, compound statement, table element, create a domain or type alias, view element, GRANT, is distinct, is null predicate, match predicate, boolean factor, procedure parameter, procedure result column, like regex predicate, in predicate, temporary table element

Keyword	Usage
NULL	alter column options, table element, create a domain or type alias, view element, is null predicate, non numeric literal, procedure parameter, procedure result column, temporary table element, xml query
OF	alter, ALTER TRIGGER, create trigger
OFFSET	limit clause
ON	alter, ALTER TRIGGER, create foreign temp table, CREATE POLICY, create temporary table, create trigger, DROP POLICY, GRANT, loop statement, qualified table, Revoke GRANT, xml query
ONLY	fetch clause
OPTION	option clause
OPTIONS	alter child options list, alter options list, options clause
OR	boolean value expression
ORDER	GRANT, order by clause
OUT	procedure parameter
OUTER	qualified table
OVER	window specification
PARAMETER	ALTER PROCEDURE
PARTITION	window specification
PERCENT_RANK	analytic aggregate function
PRIMARY	create temporary table, inline constraint, primary key
PROCEDURE	alter, ALTER PROCEDURE, create procedure, CREATE POLICY, DROP POLICY, drop procedure, GRANT, Revoke GRANT
RANGE	window frame
REAL	simple data type

Keyword	Usage
REFERENCES	foreign key
RETURN	assignment statement, return statement, data statement
RETURNS	create procedure
REVOKE	Revoke GRANT
RIGHT	function, qualified table
ROLLUP	group by clause
ROW	array table, fetch clause, for each row trigger action, limit clause, text table, window frame bound
ROWS	array table, create temporary table, fetch clause, limit clause, window frame
SECOND	function
SELECT	CREATE POLICY, grant type, select clause
SERVER	ALTER SERVER, create schema, create server, drop server, Import foreign schema
SET	add set child option, add set option, option namespace, update statement, set schema
SHORT	simple data type
SIMILAR	match predicate
SMALLINT	simple data type
SOME	standard aggregate function, quantified comparison predicate
SQLEXCEPTION	sql exception
SQLSTATE	sql exception
SQLWARNING	raise statement
STRING	dynamic data statement, simple data type, xml serialize

Keyword	Usage
TABLE	ALTER TABLE , create procedure, create foreign or global temporary table, create foreign temp table, create temporary table, drop table, drop table, GRANT, query primary, Revoke GRANT, table subquery
TEMPORARY	create foreign or global temporary table, create foreign temp table, create temporary table, drop table, GRANT, Revoke GRANT
THEN	case expression, searched case expression
TIME	non numeric literal, simple data type
TIMESTAMP	non numeric literal, simple data type
TINYINT	simple data type
TO	rename column options, RENAME Table, CREATE POLICY, DROP POLICY, GRANT, match predicate
TRAILING	function
TRANSLATE	function
TRIGGER	alter, ALTER TRIGGER , create trigger
TRUE	explain option, json table, non numeric literal
UNION	cross join, query expression body
UNIQUE	other constraints, inline constraint
UNKNOWN	non numeric literal
UPDATE	alter, ALTER TRIGGER , CREATE POLICY, create trigger, dynamic data statement, grant type, update statement
USER	function
USING	CREATE POLICY, dynamic data statement
VALUES	query primary

Keyword	Usage
VARBINARY	simple data type, xml serialize
VARCHAR	simple data type, xml serialize
VIRTUAL	ALTER PROCEDURE , ALTER TABLE , create procedure, create schema, create view, drop procedure, drop schema, drop table
WHEN	case expression, searched case expression
WHERE	filter clause, where clause
WHILE	while statement
WITH	assignment statement, create role , Import another Database, query expression, data statement
WITHIN	function
WITHOUT	assignment statement, data statement
WRAPPER	ALTER DATA WRAPPER , create data wrapper, create server, Drop data wrapper
XML	explain option, simple data type
XMLAGG	ordered aggregate function
XMLATTRIBUTES	xml attributes
XMLCAST	unescapedFunction
XMLCOMMENT	function
XMLCONCAT	function
XMLELEMENT	xml element
XMLEXISTS	xml query
XMLFOREST	xml forest
XMLNAMESPACES	xml namespaces
XMLPARSE	xml parse

Keyword	Usage
XMLPI	function
XMLQUERY	xml query
XMLSERIALIZE	xml serialize
XMLTABLE	xml table
XMLTEXT	function
YEAR	function

12.2. NON-RESERVED KEYWORDS

Name	Usage
ACCESS	basicNonReserved, Import another Database
ACCESSPATTERN	basicNonReserved, other constraints
AFTER	alter, basicNonReserved, create trigger
ANALYZE	basicNonReserved, explain option
ARRAYTABLE	array table, basicNonReserved
AUTO_INCREMENT	alter column options, basicNonReserved, table element, view element
AVG	standard aggregate function, basicNonReserved
CHAIN	basicNonReserved, sql exception
COLUMNS	array table, basicNonReserved, json table, object table, text table, xml table
CONDITION	basicNonReserved, GRANT, Revoke GRANT
CONTENT	basicNonReserved, xml parse, xml serialize
CONTROL	basicNonReserved, Import another Database
COUNT	standard aggregate function, basicNonReserved

Name	Usage
COUNT_BIG	standard aggregate function, basicNonReserved
CURRENT	basicNonReserved, window frame bound
DATA	ALTER DATA WRAPPER, basicNonReserved, create data wrapper, create server, Drop data wrapper
DATABASE	ALTER DATABASE, basicNonReserved, create database, Import another Database, use database
DEFAULT	xml namespace element, non-reserved identifier, object table column, post create column, procedure parameter, xml table column
DELIMITER	basicNonReserved, text aggreate function, text table
DENSE_RANK	analytic aggregate function, basicNonReserved
DISABLED	alter, ALTER TRIGGER, basicNonReserved
DOCUMENT	basicNonReserved, xml parse, xml serialize
DOMAIN	basicNonReserved, create a domain or type alias
EMPTY	basicNonReserved, xml query
ENABLED	alter, ALTER TRIGGER, basicNonReserved
ENCODING	basicNonReserved, text aggreate function, xml serialize
EPOCH	basicNonReserved, function
EVERY	standard aggregate function, basicNonReserved
EXCEPTION	compound statement, declare statement, non-reserved identifier
EXCLUDING	basicNonReserved, xml serialize
EXPLAIN	basicNonReserved, explain
EXTRACT	basicNonReserved, function
FIRST	basicNonReserved, fetch clause, sort specification

Name	Usage
FOLLOWING	basicNonReserved, window frame bound
FORMAT	basicNonReserved, explain option
GEOGRAPHY	non-reserved identifier, simple data type
GEOMETRY	non-reserved identifier, simple data type
HEADER	basicNonReserved, text aggreate function, text table column, text table
INCLUDING	basicNonReserved, xml serialize
INDEX	other constraints, inline constraint, non-reserved identifier
INSTEAD	alter, ALTER TRIGGER, basicNonReserved, create trigger
JAAS	basicNonReserved, with role
JSON	non-reserved identifier, simple data type
JSONARRAY_AGG	basicNonReserved, ordered aggregate function
JSONOBJECT	basicNonReserved, json object
JSONTABLE	basicNonReserved, json table
KEY	basicNonReserved, create temporary table, foreign key, inline constraint, primary key
LAST	basicNonReserved, sort specification
LISTAGG	basicNonReserved, function
MASK	basicNonReserved, GRANT, Revoke GRANT
MAX	standard aggregate function, basicNonReserved, make dep options
MIN	standard aggregate function, basicNonReserved
NAME	basicNonReserved, function, xml element

Name	Usage
NAMESPACE	basicNonReserved, option namespace
NEXT	basicNonReserved, fetch clause
NONE	basicNonReserved
NULLS	basicNonReserved, sort specification
OBJECT	non-reserved identifier, simple data type
OBJECTTABLE	basicNonReserved, object table
ORDINALITY	basicNonReserved, json table column, text table column, xml table column
PASSING	basicNonReserved, object table, xml query, xml query, xml table
PATH	basicNonReserved, json table column, xml table column
POLICY	basicNonReserved, CREATE POLICY, DROP POLICY
POSITION	basicNonReserved, function
PRECEDING	basicNonReserved, window frame bound
PRESERVE	basicNonReserved, create temporary table
PRIVILEGES	basicNonReserved, GRANT, Revoke GRANT
QUARTER	basicNonReserved, function
QUERYSTRING	basicNonReserved, querystring function
QUOTE	basicNonReserved, text aggreate function, text table
RAISE	basicNonReserved, raise statement
RANK	analytic aggregate function, basicNonReserved
RENAME	ALTER PROCEDURE , ALTER TABLE , basicNonReserved
REPOSITORY	basicNonReserved, Import foreign schema

Name	Usage
RESULT	basicNonReserved, procedure parameter
ROLE	basicNonReserved, create role , drop role , with role
ROW_NUMBER	analytic aggregate function, basicNonReserved
SCHEMA	basicNonReserved, create schema, drop schema, GRANT, Import foreign schema, Revoke GRANT, set schema
SELECTOR	basicNonReserved, text table column, text table
SERIAL	alter column options, table element, view element, non-reserved identifier, temporary table element
SKIP	basicNonReserved, text table
SQL_TSI_DAY	basicNonReserved, time interval
SQL_TSI_FRAC_SECOND	basicNonReserved, time interval
SQL_TSI_HOUR	basicNonReserved, time interval
SQL_TSI_MINUTE	basicNonReserved, time interval
SQL_TSI_MONTH	basicNonReserved, time interval
SQL_TSI_QUARTER	basicNonReserved, time interval
SQL_TSI_SECOND	basicNonReserved, time interval
SQL_TSI_WEEK	basicNonReserved, time interval
SQL_TSI_YEAR	basicNonReserved, time interval
STDDEV_POP	standard aggregate function, basicNonReserved
STDDEV_SAMP	standard aggregate function, basicNonReserved
SUBSTRING	basicNonReserved, function
SUM	standard aggregate function, basicNonReserved
TEXT	basicNonReserved, explain option

Name	Usage
TEXTAGG	basicNonReserved, text aggreate function
TEXTTABLE	basicNonReserved, text table
TIMESTAMPADD	basicNonReserved, function
TIMESTAMPDIFF	basicNonReserved, function
TO_BYTES	basicNonReserved, function
TO_CHARS	basicNonReserved, function
TRANSLATOR	ALTER DATA WRAPPER , basicNonReserved, create data wrapper, create server, Drop data wrapper
TRIM	basicNonReserved, function, text table column, text table
TYPE	alter column options, basicNonReserved, create data wrapper, create server
UNBOUNDED	basicNonReserved, window frame bound
UPSERT	basicNonReserved, insert statement
USAGE	basicNonReserved, GRANT, Revoke GRANT
USE	basicNonReserved, use database
VARIADIC	basicNonReserved, procedure parameter
VAR_POP	standard aggregate function, basicNonReserved
VAR_SAMP	standard aggregate function, basicNonReserved
VERSION	basicNonReserved, create database, create server, Import another Database, use database, xml serialize
VIEW	alter, ALTER TABLE , basicNonReserved, create view, drop table
WELLFORMED	basicNonReserved, xml parse
WIDTH	basicNonReserved, text table column

Name	Usage
XMLDECLARATION	basicNonReserved, xml serialize
YAML	basicNonReserved, explain option

12.3. RESERVED KEYWORDS FOR FUTURE USE

ALLOCATE	ARE	ASENSITIVE
ASYMETRIC	AUTHORIZATION	BINARY
CALLED	CASCADED	CHARACTER
CHECK	CLOSE	COLLATE
CONNECT	CORRESPONDING	CRITERIA
CURRENT_USER	CURSOR	CYCLE
DATALINK	DEALLOCATE	DEC
DEREF	DESCRIBE	DETERMINISTIC
DISCONNECT	DLNEWCOPY	DLPREVIOUSCOPY
DLURLCOMPLETE	DLURLCOMPLETEONLY	DLURLCOMPLETEWRITE
DLURLPATH	DLURLPATHONLY	DLURLPATHWRITE
DLURLSCHEME	DLURLSERVER	DLVALUE
DYNAMIC	ELEMENT	EXTERNAL
FREE	GET	HAS
HOLD	IDENTITY	INDICATOR
INPUT	INSENSITIVE	INT
INTERVAL	ISOLATION	LARGE
LOCALTIME	LOCALTIMESTAMP	MATCH
MEMBER	METHOD	MODIFIES

MODULE	MULTISET	NATIONAL
NATURAL	NCHAR	NCLOB
NEW	NUMERIC	OLD
OPEN	OUTPUT	OVERLAPS
PRECISION	PREPARE	READS
RECURSIVE	REFERENCING	RELEASE
ROLLBACK	SAVEPOINT	SCROLL
SEARCH	SENSITIVE	SESSION_USER
SPECIFIC	SPECIFICTYPE	SQL
START	STATIC	SUBMULTILIST
SYMETRIC	SYSTEM	SYSTEM_USER
TIMEZONE_HOUR	TIMEZONE_MINUTE	TRANSLATION
TREAT	VALUE	VARYING
WHENEVER	WINDOW	XMLBINARY
XMLDOCUMENT	XMLITERATE	XMLVALIDATE

12.4. TOKENS

Name	Definition	Usage
all in group identifier	<identifier> <period> <star>	all in group
binary string literal	"X" | "x" "\'" (<hexit> <hexit>)+ "\'"	non numeric literal
colon	":"	make dep options, statement

<table>
<tr><th>Name</th><th>Definition</th><th>Usage</th></tr>
<tr><td>comma</td><td>","</td><td>alter child options list, alter options list, ARRAY expression constructor, column list, create procedure, typed element list, CREATE POLICY, create table body, create temporary table, create view body, derived column list, sql exception, named parameter list, explain, expression list, from clause, function, GRANT, identifier list, json table, limit clause, nested expression, object table, option clause, options clause, order by clause, simple data type, query expression, query primary, querystring function, Revoke GRANT, select clause, set clause list, in predicate, text aggregate function, text table, xml attributes, xml element, xml query, xml forest, xml namespaces, xml query, xml table</td></tr>
<tr><td>concat_op</td><td>"||"</td><td>common value expression</td></tr>
<tr><td>decimal numeric literal</td><td>(<digit>)* <period> <unsigned integer literal></td><td>unsigned numeric literal</td></tr>
<tr><td>digit</td><td>\["0"\-"9"\]</td><td></td></tr>
<tr><td>dollar</td><td>"$"</td><td>parameter reference</td></tr>
<tr><td>double_amp_op</td><td>"&&"</td><td>common value expression</td></tr>
<tr><td>eq</td><td>"="</td><td>assignment statement, callable statement, declare statement, named parameter list, comparison operator, set clause list</td></tr>
<tr><td>escaped function</td><td>"{" "fn"</td><td>unsigned value expression primary</td></tr>
<tr><td>escaped join</td><td>"{" "oj"</td><td>table reference</td></tr>
<tr><td>escaped type</td><td>"{" ("d" | "t" | "ts" | "b")</td><td>non numeric literal</td></tr>
<tr><td>approximate numeric literal</td><td><digit> <period> <unsigned integer literal> \["e","E"\] (<plus> | <minus>)? <unsigned integer literal></td><td>unsigned numeric literal</td></tr>
</table>

Name	Definition	Usage
ge	">="	comparison operator
gt	">"	named parameter list, comparison operator
hexit	\["a"\-"f","A"\-"F"\] | <digit>	
identifier	<quoted_id> (<period> <quoted_id>)*	create a domain or type alias, identifier, data type, Unqualified identifier, unsigned value expression primary
id_part	("" | "@" | "#" | <letter>) (<letter> | "" | <digit>)*	
lbrace	"{"	callable statement, match predicate
le	"⇐"	comparison operator
letter	\["a"\-"z","A"\-"Z"\] | \["\u0153"\-"\ufffd"\]	

Name	Definition	Usage
lparen	"("	standard aggregate function, alter child options list, alter options list, analytic aggregate function, ARRAY expression constructor, array table, callable statement, column list, other constraints, create procedure, CREATE POLICY, create table body, create temporary table, create view body, explain, filter clause, function, group by clause, if statement, json object, json table, loop statement, make dep options, nested expression, object table, options clause, ordered aggregate function, simple data type, query primary, querystring function, in predicate, call statement, subquery, quantified comparison predicate, table subquery, table primary, text aggreate function, text table, unescapedFunction, while statement, window specification, with list element, xml attributes, xml element, xml query, xml forest, xml namespaces, xml parse, xml query, xml serialize, xml table
lsbrace	"["	ARRAY expression constructor, basic data type, data type, value expression primary
lt	"<"	comparison operator
minus	"-"	plus or minus
ne	"<>"	comparison operator
ne2	"!="	comparison operator
period	"."	
plus	"+"	plus or minus
qmark	"?"	callable statement, parameter reference

Name	Definition	Usage
quoted_id	<id_part> | "\"" ("\"\"" | ~\["\""\])+ "\""	
rbrace	"}"	callable statement, match predicate, non numeric literal, table reference, unsigned value expression primary
rparen	")"	standard aggregate function, alter child options list, alter options list, analytic aggregate function, ARRAY expression constructor, array table, callable statement, column list, other constraints, create procedure, CREATE POLICY, create table body, create temporary table, create view body, explain, filter clause, function, group by clause, if statement, json object, json table, loop statement, make dep options, nested expression, object table, options clause, ordered aggregate function, simple data type, query primary, querystring function, in predicate, call statement, subquery, quantified comparison predicate, table subquery, table primary, text aggreate function, text table, unescapedFunction, while statement, window specification, with list element, xml attributes, xml element, xml query, xml forest, xml namespaces, xml parse, xml query, xml serialize, xml table
rsbrace	"]"	ARRAY expression constructor, basic data type, data type, value expression primary
semicolon	";"	delimited statement
slash	"/"	star or slash
star	"*"	standard aggregate function, dynamic data statement, select clause, star or slash

Name	Definition	Usage
string literal	("N" | "E")? "\'" ("\'\'" | ~\["\'"\])* "\'"	string
unsigned integer literal	(<digit>)+	unsigned integer, unsigned numeric literal

12.5. PRODUCTION CROSS-REFERENCE

Name	Usage
add set child option	alter child options list
add set option	alter options list
standard aggregate function	unescapedFunction
all in group	select sublist
alter	directly executable statement
ADD column	ALTER TABLE
ADD constraint	ALTER TABLE
alter child option pair	add set child option
alter child options list	alter column options
alter column options	ALTER PROCEDURE , ALTER TABLE
ALTER DATABASE	alterStatement
DROP column	ALTER TABLE
alter option pair	add set option
alter options list	ALTER DATABASE , ALTER PROCEDURE , ALTER SERVER , ALTER TABLE , ALTER DATA WRAPPER
ALTER PROCEDURE	alterStatement
rename column options	ALTER PROCEDURE , ALTER TABLE
RENAME Table	ALTER TABLE

Name	Usage
ALTER SERVER	alterStatement
alterStatement	ddl statement
ALTER TABLE	alterStatement
ALTER DATA WRAPPER	alterStatement
ALTER TRIGGER	alterStatement
analytic aggregate function	unescapedFunction
ARRAY expression constructor	unsigned value expression primary
array table	table primary
assignment statement	delimited statement
assignment statement operand	assignment statement, declare statement
basicNonReserved	create a domain or type alias, non-reserved identifier, data type
between predicate	boolean primary
boolean primary	CREATE POLICY, filter clause, boolean factor
branching statement	delimited statement
callable statement	
case expression	unsigned value expression primary
character	match predicate, text aggregate function, text table
column list	other constraints, create temporary table, foreign key, insert statement, primary key, with list element
common value expression	between predicate, boolean primary, comparison predicate, sql exception, function, is distinct, match predicate, like regex predicate, in predicate, text table
comparison predicate	boolean primary

Name	Usage
boolean term	boolean value expression
boolean value expression	condition
compound statement	statement, directly executable statement
other constraints	table constraint
table element	ADD column, create table body
create procedure	ddl statement
create data wrapper	ddl statement
create database	ddl statement
create a domain or type alias	ddl statement
typed element list	array table, dynamic data statement
create foreign or global temporary table	create table
create foreign temp table	directly executable statement
option namespace	ddl statement
CREATE POLICY	ddl statement
create role	ddl statement
create schema	ddl statement
create server	ddl statement
create table	ddl statement
create table body	create foreign or global temporary table, create foreign temp table
create temporary table	directly executable statement
create trigger	ddl statement, directly executable statement
create view	create table

Name	Usage
create view body	create view
view element	create view body
condition	expression, having clause, if statement, qualified table, searched case expression, where clause, while statement
cross join	joined table
ddl statement	ddl statement
declare statement	delimited statement
delete statement	assignment statement operand, directly executable statement
delimited statement	statement
derived column	derived column list, object table, querystring function, text aggregate function, xml attributes, xml query, xml query, xml table
derived column list	json object, xml forest
drop option	alter child options list
Drop data wrapper	ddl statement
drop option	alter options list
DROP POLICY	ddl statement
drop procedure	ddl statement
drop role	ddl statement
drop schema	ddl statement
drop server	ddl statement
drop table	directly executable statement
drop table	ddl statement

Name	Usage
dynamic data statement	data statement
raise error statement	delimited statement
sql exception	assignment statement operand, exception reference
exception reference	sql exception, raise statement
named parameter list	callable statement, call statement
exists predicate	boolean primary
explain	
explain option	explain
expression	standard aggregate function, ARRAY expression constructor, assignment statement operand, case expression, derived column, dynamic data statement, raise error statement, named parameter list, expression list, function, nested expression, object table column, ordered aggregate function, post create column, procedure parameter, querystring function, return statement, searched case expression, select derived column, set clause list, sort key, quantified comparison predicate, unescapedFunction, xml table column, xml element, xml parse, xml serialize
expression list	callable statement, other constraints, function, group by clause, query primary, call statement, window specification
fetch clause	limit clause
filter clause	function, unescapedFunction
for each row trigger action	alter, ALTER TRIGGER, create trigger
foreign key	table constraint
from clause	query
function	unescapedFunction, unsigned value expression primary
GRANT	ddl statement

Name	Usage
group by clause	query
having clause	query
identifier	alter, alter child option pair, alter column options, ALTER DATABASE, DROP column, alter option pair, ALTER PROCEDURE, rename column options, RENAME Table, ALTER SERVER, ALTER TABLE, ALTER DATA WRAPPER, ALTER TRIGGER, array table, assignment statement, branching statement, callable statement, column list, compound statement, table element, create data wrapper, create database, typed element list, create foreign temp table, option namespace, CREATE POLICY, create schema, create trigger, view element, declare statement, delete statement, derived column, drop option, Drop data wrapper, drop option, DROP POLICY, drop procedure, drop role, drop schema, drop server, drop table, drop table, dynamic data statement, exception reference, named parameter list, foreign key, function, GRANT, identifier list, Import another Database, Import foreign schema, insert statement, into clause, json table column, json table, loop statement, xml namespace element, object table column, object table, option clause, option pair, procedure parameter, procedure result column, query primary, Revoke GRANT, select derived column, set clause list, statement, call statement, table subquery, table constraint, temporary table element, text aggreate function, text table column, text table, table name, update statement, use database, set schema, with list element, xml table column, xml element, xml serialize, xml table
identifier list	create schema, with role
if statement	statement
Import another Database	ddl statement
Import foreign schema	ddl statement
inline constraint	post create column
insert statement	assignment statement operand, directly executable statement
integer parameter	fetch clause, limit clause

Name	Usage
unsigned integer	dynamic data statement, function, GRANT, integer parameter, make dep options, parameter reference, simple data type, text table column, text table, window frame bound
time interval	function
into clause	query
is distinct	boolean primary
is null predicate	boolean primary
joined table	table primary, table reference
json table column	json table
json object	function
json table	table primary
limit clause	query expression body
loop statement	statement
make dep options	option clause, table primary
match predicate	boolean primary
xml namespace element	xml namespaces
nested expression	unsigned value expression primary
non numeric literal	alter child option pair, alter option pair, option pair, value expression primary
non-reserved identifier	identifier, Unqualified identifier, unsigned value expression primary
boolean factor	boolean term
object table column	object table
object table	table primary

Name	Usage
comparison operator	comparison predicate, quantified comparison predicate
option clause	callable statement, delete statement, insert statement, query expression body, call statement, update statement
option pair	options clause
options clause	create procedure, create data wrapper, create database, create schema, create server, create table body, create view, create view body, Import foreign schema, post create column, procedure parameter, procedure result column, table constraint
order by clause	function, ordered aggregate function, query expression body, text aggreate function, window specification
ordered aggregate function	unescapedFunction
parameter reference	unsigned value expression primary
basic data type	typed element list, json table column, object table column, data type, temporary table element, text table column, xml table column
data type	alter column options, table element, create procedure, create a domain or type alias, view element, declare statement, function, procedure parameter, procedure result column, unescapedFunction
simple data type	basic data type
numeric value expression	common value expression, value expression primary
plus or minus	alter child option pair, alter option pair, option pair, numeric value expression, value expression primary
post create column	table element, view element
primary key	table constraint
procedure parameter	create procedure
procedure result column	create procedure

Name	Usage
qualified table	joined table
query	query primary
query expression	alter, ALTER TABLE , ARRAY expression constructor, assignment statement operand, create view, insert statement, loop statement, subquery, table subquery, directly executable statement, with list element
query expression body	query expression, query primary
query primary	query term
querystring function	function
query term	query expression body
raise statement	delimited statement
grant type	GRANT, Revoke GRANT
with role	create role
like regex predicate	boolean primary
return statement	delimited statement
Revoke GRANT	ddl statement
searched case expression	unsigned value expression primary
select clause	query
select derived column	select sublist
select sublist	select clause
set clause list	dynamic data statement, update statement
in predicate	boolean primary
sort key	sort specification
sort specification	order by clause

Name	Usage
data statement	delimited statement
statement	alter, ALTER PROCEDURE , compound statement, create procedure, for each row trigger action, if statement, loop statement, while statement
call statement	assignment statement, subquery, table subquery, directly executable statement
string	character, create database, option namespace, create server, function, GRANT, Import another Database, json table column, json table, xml namespace element, non numeric literal, object table column, object table, text table column, text table, use database, xml table column, xml query, xml query, xml serialize, xml table
subquery	exists predicate, in predicate, quantified comparison predicate, unsigned value expression primary
quantified comparison predicate	boolean primary
table subquery	table primary
table constraint	ADD constraint, create table body, create view body
temporary table element	create temporary table
table primary	cross join, joined table
table reference	from clause, qualified table
text aggregate function	unescapedFunction
text table column	text table
text table	table primary
term	numeric value expression
star or slash	term
table name	table primary
unescapedFunction	unsigned value expression primary

Name	Usage
Unqualified identifier	create procedure, create data wrapper, create foreign or global temporary table, create foreign temp table, create role , create server, create temporary table, create view
unsigned numeric literal	alter child option pair, alter option pair, option pair, value expression primary
unsigned value expression primary	integer parameter, value expression primary
update statement	assignment statement operand, directly executable statement
use database	ddl statement
set schema	ddl statement
directly executable statement	explain, data statement
value expression primary	array table, json table, term
where clause	delete statement, query, update statement
while statement	statement
window frame	window specification
window frame bound	window frame
window specification	unescapedFunction
with list element	query expression
xml attributes	xml element
xml table column	xml table
xml element	function
xml query	boolean primary
xml forest	function
xml namespaces	xml element, xml query, xml forest, xml query, xml table

Name	Usage
xml parse	function
xml query	function
xml serialize	function
xml table	table primary

12.6. PRODUCTIONS

12.6.1. *string* ::=

- <string literal>

A string literal value. Use '' to escape ' in the string.

Example:

> 'a string'

> 'it''s a string'

12.6.2. *non-reserved identifier* ::=

- EXCEPTION
- SERIAL
- OBJECT
- INDEX
- JSON
- GEOMETRY
- GEOGRAPHY
- DEFAULT
- <basicNonReserved>

Allows non-reserved keywords to be parsed as identifiers

Example: SELECT **COUNT** FROM ...

12.6.3. *basicNonReserved* ::=

- INSTEAD

- VIEW
- ENABLED
- DISABLED
- KEY
- TEXTAGG
- COUNT
- COUNT_BIG
- ROW_NUMBER
- RANK
- DENSE_RANK
- SUM
- AVG
- MIN
- MAX
- EVERY
- STDDEV_POP
- STDDEV_SAMP
- VAR_SAMP
- VAR_POP
- DOCUMENT
- CONTENT
- TRIM
- EMPTY
- ORDINALITY
- PATH
- FIRST
- LAST
- NEXT
- SUBSTRING

- EXTRACT
- TO_CHARS
- TO_BYTES
- TIMESTAMPADD
- TIMESTAMPDIFF
- QUERYSTRING
- NAMESPACE
- RESULT
- ACCESSPATTERN
- AUTO_INCREMENT
- WELLFORMED
- SQL_TSI_FRAC_SECOND
- SQL_TSI_SECOND
- SQL_TSI_MINUTE
- SQL_TSI_HOUR
- SQL_TSI_DAY
- SQL_TSI_WEEK
- SQL_TSI_MONTH
- SQL_TSI_QUARTER
- SQL_TSI_YEAR
- TEXTTABLE
- ARRAYTABLE
- JSONTABLE
- SELECTOR
- SKIP
- WIDTH
- PASSING
- NAME
- ENCODING

- COLUMNS
- DELIMITER
- QUOTE
- HEADER
- NULLS
- OBJECTTABLE
- VERSION
- INCLUDING
- EXCLUDING
- XMLDECLARATION
- VARIADIC
- RAISE
- CHAIN
- JSONARRAY_AGG
- JSONOBJECT
- PRESERVE
- UPSERT
- AFTER
- TYPE
- TRANSLATOR
- JAAS
- CONDITION
- MASK
- ACCESS
- CONTROL
- NONE
- DATA
- DATABASE
- PRIVILEGES

- ROLE
- SCHEMA
- USE
- REPOSITORY
- RENAME
- DOMAIN
- USAGE
- POSITION
- CURRENT
- UNBOUNDED
- PRECEDING
- FOLLOWING
- LISTAGG
- EXPLAIN
- ANALYZE
- TEXT
- FORMAT
- YAML
- EPOCH
- QUARTER
- POLICY

12.6.4. *Unqualified identifier* ::=

- <identifier>
- <non-reserved identifier>

Unqualified name of a single entity.

Example:

```
"tbl"
```

12.6.5. *identifier* ::=

- <identifier>
- <non-reserved identifier>

Partial or full name of a single entity.

Example:

```
tbl.col
```

```
"tbl"."col"
```

12.6.6. *create trigger* ::=

- CREATE TRIGGER (<identifier>)? ON <identifier> ((INSTEAD OF) | AFTER) (INSERT | UPDATE | DELETE) AS <for each row trigger action>

Creates a trigger action on the given target.

Example:

```
CREATE TRIGGER ON vw INSTEAD OF INSERT AS FOR EACH ROW BEGIN ATOMIC ... END
```

12.6.7. *alter* ::=

- ALTER ((VIEW <identifier> AS <query expression>) | (PROCEDURE <identifier> AS <statement>) | (TRIGGER (<identifier>)? ON <identifier> ((INSTEAD OF) | AFTER) (INSERT | UPDATE | DELETE) ((AS <for each row trigger action>) | ENABLED | DISABLED)))

Alter the given target.

Example:

```
ALTER VIEW vw AS SELECT col FROM tbl
```

12.6.8. *for each row trigger action* ::=

- FOR EACH ROW ((BEGIN (ATOMIC)? (<statement>)* END) | <statement>)

Defines an action to perform on each row.

Example:

```
FOR EACH ROW BEGIN ATOMIC ... END
```

12.6.9. *explain* ::=

- EXPLAIN (<lparen> <explain option> (<comma> <explain option>)* <rparen>)? <directly executable statement>

Returns the query plan for the statement

Example: EXPLAIN select 1

12.6.10. *explain option* ::=

- (ANALYZE (TRUE | FALSE)?)
- (FORMAT (XML | TEXT | YAML)?)

Option for the explain statement

Example: FORMAT YAML

12.6.11. *directly executable statement* ::=

- <query expression>
- <call statement>
- <insert statement>
- <update statement>
- <delete statement>
- <drop table>
- <create temporary table>
- <create foreign temp table>
- <alter>
- <create trigger>
- <compound statement>

A statement that can be executed at runtime.

Example:

```
SELECT * FROM tbl
```

12.6.12. *drop table* ::=

- DROP TABLE <identifier>

Drop the given table.

Example:

```
DROP TABLE #temp
```

12.6.13. *create temporary table* ::=

- CREATE (LOCAL)? TEMPORARY TABLE <Unqualified identifier> <lparen> <temporary table element> (<comma> <temporary table element>)* (<comma> PRIMARY KEY <column list>)? <rparen> (ON COMMIT PRESERVE ROWS)?

Creates a temporary table.

Example:

```
CREATE LOCAL TEMPORARY TABLE tmp (col integer)
```

12.6.14. *temporary table element* ::=

- <identifier> (<basic data type> | SERIAL) (NOT NULL)?

Defines a temporary table column.

Example:

```
col string NOT NULL
```

12.6.15. *raise error statement* ::=

- ERROR <expression>

Raises an error with the given message.

Example:

```
ERROR 'something went wrong'
```

12.6.16. *raise statement* ::=

- RAISE (SQLWARNING)? <exception reference>

Raises an error or warning with the given message.

Example:

```
RAISE SQLEXCEPTION 'something went wrong'
```

12.6.17. *exception reference* ::=

- <identifier>
- <sql exception>

a reference to an exception

Example:

```
SQLEXCEPTION 'something went wrong' SQLSTATE '00X', 2
```

12.6.18. *sql exception* ::=

- SQLEXCEPTION <common value expression> (SQLSTATE <common value expression> (<comma> <common value expression>)?)? (CHAIN <exception reference>)?

creates a sql exception or warning with the specified message, state, and code

Example:

```
SQLEXCEPTION 'something went wrong' SQLSTATE '00X', 2
```

12.6.19. *statement* ::=

- ((<identifier> <colon>)? (<loop statement> | <while statement> | <compound statement>))
- <if statement> | <delimited statement>

A procedure statement.

Example:

```
IF (x = 5) BEGIN ... END
```

12.6.20. *delimited statement* ::=

- (<assignment statement> | <data statement> | <raise error statement> | <raise statement> | <declare statement> | <branching statement> | <return statement>) <semicolon>

A procedure statement terminated by ;.

Example:

```
SELECT * FROM tbl;
```

12.6.21. *compound statement* ::=

- BEGIN ((NOT)? ATOMIC)? (<statement>)* (EXCEPTION <identifier> (<statement>)*)? END

A procedure statement block contained in BEGIN END.

Example:

```
BEGIN NOT ATOMIC ... END
```

12.6.22. *branching statement* ::=

- ((BREAK | CONTINUE) (<identifier>)?)
- (LEAVE <identifier>)

A procedure branching control statement, which typically specifies a label to return control to.

Example:

```
BREAK x
```

12.6.23. *return statement* ::=

- RETURN (<expression>)?

A return statement.

Example:

```
RETURN 1
```

12.6.24. *while statement* ::=

- WHILE <lparen> <condition> <rparen> <statement>

A procedure while statement that executes until its condition is false.

Example:

```
WHILE (var) BEGIN ... END
```

12.6.25. *loop statement* ::=

- LOOP ON <lparen> <query expression> <rparen> AS <identifier> <statement>

A procedure loop statement that executes over the given cursor.

Example:

```
LOOP ON (SELECT * FROM tbl) AS x BEGIN ... END
```

12.6.26. *if statement* ::=

- IF <lparen> <condition> <rparen> <statement> (ELSE <statement>)?

A procedure loop statement that executes over the given cursor.

Example:

```
IF (boolVal) BEGIN variables.x = 1 END ELSE BEGIN variables.x = 2 END
```

12.6.27. *declare statement* ::=

- DECLARE (<data type> | EXCEPTION) <identifier> (<eq> <assignment statement operand>)?

A procedure declaration statement that creates a variable and optionally assigns a value.

Example:

```
DECLARE STRING x = 'a'
```

12.6.28. *assignment statement* ::=

- <identifier> <eq> (<assignment statement operand> | (<call statement> ((WITH | WITHOUT) RETURN)?))

Assigns a variable a value in a procedure.

Example:

```
x = 'b'
```

12.6.29. *assignment statement operand* ::=

- <insert statement>
- <update statement>
- <delete statement>
- <expression>
- <query expression>
- <sql exception>

A value or command that can be used in an assignment. {note}All assignments except for expression are deprecated.{note}

12.6.30. *data statement* ::=

- (<directly executable statement> | <dynamic data statement>) ((WITH | WITHOUT) RETURN)?

A procedure statement that executes a SQL statement. An update statement can have its update count accessed via the ROWCOUNT variable.

12.6.31. *dynamic data statement* ::=

- (EXECUTE | EXEC) (STRING | IMMEDIATE)? <expression> (AS <typed element list> (INTO <identifier>)?)? (USING <set clause list>)? (UPDATE (<unsigned integer> | <star>))?

A procedure statement that can execute arbitrary sql.

Example:

```
EXECUTE IMMEDIATE 'SELECT * FROM tbl' AS x STRING INTO #temp
```

12.6.32. *set clause list* ::=

- <identifier> <eq> <expression> (<comma> <identifier> <eq> <expression>)*

A list of value assignments.

Example:

```
col1 = 'x', col2 = 'y' ...
```

12.6.33. *typed element list* ::=

- <identifier> <basic data type> (<comma> <identifier> <basic data type>)*

A list of typed elements.

Example:

```
col1 string, col2 integer ...
```

12.6.34. *callable statement* ::=

- <lbrace> (<qmark> <eq>)? CALL <identifier> (<lparen> (<named parameter list> | (<expression list>)?) <rparen>)? <rbrace> (<option clause>)?

A callable statement defined using JDBC escape syntax.

Example:

```
{? = CALL proc}
```

12.6.35. *call statement* ::=

- ((EXEC | EXECUTE | CALL) <identifier> <lparen> (<named parameter list> | (<expression list>)?) <rparen>) (<option clause>)?

Executes the procedure with the given parameters.

Example:

```
CALL proc('a', 1)
```

12.6.36. *named parameter list* ::=

- (<identifier> <eq> (<gt>)? <expression> (<comma> <identifier> <eq> (<gt>)? <expression>)*)

A list of named parameters.

Example:

```
param1 => 'x', param2 => 1
```

12.6.37. *insert statement* ::=

- (INSERT | MERGE | UPSERT) INTO <identifier> (<column list>)? <query expression> (<option clause>)?

Inserts values into the given target.

Example:

```
INSERT INTO tbl (col1, col2) VALUES ('a', 1)
```

12.6.38. *expression list* ::=

- <expression> (<comma> <expression>)*

A list of expressions.

Example:

```
col1, 'a', ...
```

12.6.39. *update statement* ::=

- UPDATE <identifier> ((AS)? <identifier>)? SET <set clause list> (<where clause>)? (<option clause>)?

Update values in the given target.

Example:

```
UPDATE tbl SET (col1 = 'a') WHERE col2 = 1
```

12.6.40. *delete statement* ::=

- DELETE FROM <identifier> ((AS)? <identifier>)? (<where clause>)? (<option clause>)?

Delete rows from the given target.

Example:

```
DELETE FROM tbl WHERE col2 = 1
```

12.6.41. *query expression* ::=

- (WITH <with list element> (<comma> <with list element>)*)? <query expression body>

A declarative query for data.

Example:

```
SELECT * FROM tbl WHERE col2 = 1
```

12.6.42. *with list element* ::=

- <identifier> (<column list>)? AS <lparen> <query expression> <rparen>

A query expression for use in the enclosing query.

Example:

```
X (Y, Z) AS (SELECT 1, 2)
```

12.6.43. *query expression body* ::=

- <query term> ((UNION | EXCEPT) (ALL | DISTINCT)? <query term>)* (<order by clause>)? (<limit clause>)? (<option clause>)?

The body of a query expression, which can optionally be ordered and limited.

Example:

```
SELECT * FROM tbl ORDER BY col1 LIMIT 1
```

12.6.44. *query term* ::=

- <query primary> (INTERSECT (ALL | DISTINCT)? <query primary>)*

Used to establish INTERSECT precedence.

Example:

```
SELECT * FROM tbl
```

```
SELECT * FROM tbl1 INTERSECT SELECT * FROM tbl2
```

12.6.45. *query primary* ::=

- <query>
- (VALUES <lparen> <expression list> <rparen> (<comma> <lparen> <expression list> <rparen>)*)
- (TABLE <identifier>)
- (<lparen> <query expression body> <rparen>)

A declarative source of rows.

Example:

```
TABLE tbl
```

```
SELECT * FROM tbl1
```

12.6.46. *query* ::=

- <select clause> (<into clause>)? (<from clause> (<where clause>)? (<group by clause>)? (<having clause>)?)?

A SELECT query.

Example:

```
SELECT col1, max(col2) FROM tbl GROUP BY col1
```

12.6.47. *into clause* ::=

- INTO <identifier>

Used to direct the query into a table. {note}This is deprecated. Use INSERT INTO with a query expression instead.{note}

Example:

```
INTO tbl
```

12.6.48. *select clause* ::=

- SELECT (ALL | DISTINCT)? (<star> | (<select sublist> (<comma> <select sublist>)*))

The columns returned by a query. Can optionally be distinct.

Example:

```
SELECT *
```

```
SELECT DISTINCT a, b, c
```

12.6.49. *select sublist* ::=

- <select derived column>
- <all in group >

An element in the select clause

Example:

```
tbl.*
```

```
tbl.col AS x
```

12.6.50. *select derived column* ::=

- (<expression> ((AS)? <identifier>)?)

A select clause item that selects a single column. {note}This is slightly different than a derived column in that the AS keyword is optional.{note}

Example:

```
tbl.col AS x
```

12.6.51. *derived column* ::=

- (<expression> (AS <identifier>)?)

An optionally named expression.

Example:

```
tbl.col AS x
```

12.6.52. *all in group* ::=

- <all in group identifier >

A select sublist that can select all columns from the given group.

Example:

```
tbl.*
```

12.6.53. *ordered aggregate function* ::=

- (XMLAGG | ARRAY_AGG | JSONARRAY_AGG) <lparen> <expression> (<order by clause >)? <rparen>

An aggregate function that can optionally be ordered.

Example:

```
XMLAGG(col1) ORDER BY col2
```

```
ARRAY_AGG(col1)
```

12.6.54. *text aggreate function* ::=

- TEXTAGG <lparen> (FOR)? <derived column> (<comma> <derived column>)* (DELIMITER <character>)? ((QUOTE <character>) | (NO QUOTE))? (HEADER)? (ENCODING <identifier>)? (<order by clause >)? <rparen>

An aggregate function for creating separated value clobs.

Example:

```
TEXTAGG (col1 as t1, col2 as t2 DELIMITER ',' HEADER)
```

12.6.55. *standard aggregate function* ::=

- ((COUNT | COUNT_BIG) <lparen> <star> <rparen>)
- ((COUNT | COUNT_BIG | SUM | AVG | MIN | MAX | EVERY | STDDEV_POP | STDDEV_SAMP | VAR_SAMP | VAR_POP | SOME | ANY) <lparen> (DISTINCT | ALL)? <expression> <rparen>)

A standard aggregate function.

Example:

```
COUNT(*)
```

12.6.56. *analytic aggregate function* ::=

- (ROW_NUMBER | RANK | DENSE_RANK | PERCENT_RANK | CUME_DIST) <lparen> <rparen>

An analytic aggregate function.

Example:

```
ROW_NUMBER()
```

12.6.57. *filter clause* ::=

- FILTER <lparen> WHERE <boolean primary> <rparen>

An aggregate filter clause applied prior to accumulating the value.

Example:

```
FILTER (WHERE col1='a')
```

12.6.58. *from clause* ::=

- FROM (<table reference> (<comma> <table reference>)*)

A query from clause containing a list of table references.

Example:

```
FROM a, b
```

```
FROM a right outer join b, c, d join e".</p>
```

12.6.59. *table reference* ::=

- (<escaped join> <joined table> <rbrace>)
- <joined table>

An optionally escaped joined table.

Example:

```
a
```

```
a inner join b
```

12.6.60. *joined table* ::=

- <table primary> (<cross join> | <qualified table>)*

A table or join.

Example:

```
a
```

```
a inner join b
```

12.6.61. *cross join* ::=

- ((CROSS | UNION) JOIN <table primary>)

A cross join.

Example:

```
a CROSS JOIN b
```

12.6.62. *qualified table* ::=

- (((RIGHT (OUTER)?) | (LEFT (OUTER)?) | (FULL (OUTER)?) | INNER)? JOIN <table reference> ON <condition>)

An INNER or OUTER join.

Example:

```
a inner join b
```

12.6.63. *table primary* ::=

- (<text table> | <array table> | <json table> | <xml table> | <object table> | <table name> | <table subquery> | (<lparen> <joined table> <rparen>)) ((MAKEDEP <make dep options>) | MAKENOTDEP)? ((MAKEIND <make dep options>))?

A single source of rows.

Example:

```
a
```

12.6.64. *make dep options* ::=

- (<lparen> (MAX <colon> <unsigned integer>)? ((NO)? JOIN)? <rparen>)?

options for the make dep hint

Example:

```
(min:10000)
```

12.6.65. *xml serialize* ::=

- XMLSERIALIZE <lparen> (DOCUMENT | CONTENT)? <expression> (AS (STRING | VARCHAR | CLOB | VARBINARY | BLOB))? (ENCODING <identifier>)? (VERSION <string>)? ((INCLUDING | EXCLUDING) XMLDECLARATION)? <rparen>

Serializes an XML value.

Example:

```
XMLSERIALIZE(col1 AS CLOB)
```

12.6.66. *array table* ::=

- ARRAYTABLE <lparen> (ROW | ROWS)? <value expression primary> COLUMNS <typed element list> <rparen> (AS)? <identifier>

The ARRAYTABLE table function creates tabular results from arrays. It can be used as a nested table reference.

Example:

```
ARRAYTABLE (col1 COLUMNS x STRING) AS y
```

12.6.67. *json table* ::=

- JSONTABLE <lparen> <value expression primary> <comma> <string> (<comma> (TRUE | FALSE))? COLUMNS <json table column> (<comma> <json table column>)* <rparen> (AS)? <identifier>

The JSONTABLE table function creates tabular results from JSON. It can be used as a nested table reference.

Example:

```
JSONTABLE (col1, '$..book', false COLUMNS x STRING) AS y
```

12.6.68. *json table column* ::=

- <identifier> ((FOR ORDINALITY) | (<basic data type> (PATH <string>)?))

json table column.

Example:

```
col FOR ORDINALITY
```

12.6.69. *text table* ::=

- TEXTTABLE <lparen> <common value expression> (SELECTOR <string>)? COLUMNS <text

table column> (<comma> <text table column>)* ((NO ROW DELIMITER) | (ROW DELIMITER <character>))? (DELIMITER <character>)? ((ESCAPE <character>) | (QUOTE <character>))? (HEADER (<unsigned integer>)?)? (SKIP <unsigned integer>)? (NO TRIM)? <rparen> (AS)? <identifier>

The TEXTTABLE table function creates tabular results from text. It can be used as a nested table reference.

Example:

```
TEXTTABLE (file COLUMNS x STRING) AS y
```

12.6.70. *text table column* ::=

- <identifier> ((FOR ORDINALITY) | ((HEADER <string>)? <basic data type> (WIDTH <unsigned integer> (NO TRIM)?)? (SELECTOR <string> <unsigned integer>)?))

A text table column.

Example:

```
x INTEGER WIDTH 6
```

12.6.71. *xml query* ::=

- XMLEXISTS <lparen> (<xml namespaces> <comma>)? <string> (PASSING <derived column> (<comma> <derived column>)*)? <rparen>

Executes an XQuery to return an XML result.

Example:

```
XMLQUERY('<a>...</a>' PASSING doc)
```

12.6.72. *xml query* ::=

- XMLQUERY <lparen> (<xml namespaces> <comma>)? <string> (PASSING <derived column> (<comma> <derived column>)*)? ((NULL | EMPTY) ON EMPTY)? <rparen>

Executes an XQuery to return an XML result.

Example:

```
XMLQUERY('<a>...</a>' PASSING doc)
```

12.6.73. *object table* ::=

- OBJECTTABLE <lparen> (LANGUAGE <string>)? <string> (PASSING <derived column> (<comma> <derived column>)*)? COLUMNS <object table column> (<comma> <object table column>)* <rparen> (AS)? <identifier>

Returns table results by processing a script.

Example:

```
OBJECTTABLE('z' PASSING val AS z COLUMNS col OBJECT 'teiid_row') AS X
```

12.6.74. *object table column* ::=

- <identifier> <basic data type> <string> (DEFAULT <expression>)?

object table column.

Example:

```
y integer 'teiid_row_number'
```

12.6.75. *xml table* ::=

- XMLTABLE <lparen> (<xml namespaces> <comma>)? <string> (PASSING <derived column> (<comma> <derived column>)*)? (COLUMNS <xml table column> (<comma> <xml table column>)*)? <rparen> (AS)? <identifier>

Returns table results by processing an XQuery.

Example:

```
XMLTABLE('/a/b' PASSING doc COLUMNS col XML PATH '.') AS X
```

12.6.76. *xml table column* ::=

- <identifier> ((FOR ORDINALITY) | (<basic data type> (DEFAULT <expression>)? (PATH <string>)?))

XML table column.

Example:

```
y FOR ORDINALITY
```

12.6.77. *unsigned integer* ::=

- <unsigned integer literal>

An unsigned interger value.

Example:

```
12345
```

12.6.78. *table subquery* ::=

- (TABLE | LATERAL)? <lparen> (<query expression> | <call statement>) <rparen> (AS)? <identifier>

A table defined by a subquery.

Example:

```
(SELECT * FROM tbl) AS x
```

12.6.79. *table name* ::=

- (<identifier> ((AS)? <identifier>)?)

A table named in the FROM clause.

Example:

```
tbl AS x
```

12.6.80. *where clause* ::=

- WHERE <condition>

Specifies a search condition

Example:

```
WHERE x = 'a'
```

12.6.81. *condition* ::=

- <boolean value expression >

A boolean expression.

12.6.82. *boolean value expression* ::=

- <boolean term> (OR <boolean term>)*

An optionally ORed boolean expression.

12.6.83. *boolean term* ::=

- <boolean factor> (AND <boolean factor>)*

An optional ANDed boolean factor.

12.6.84. *boolean factor* ::=

- (NOT)? <boolean primary>

A boolean factor.

Example:

```
NOT x = 'a'
```

12.6.85. *boolean primary* ::=

- (<common value expression> (<between predicate> | <match predicate> | <like regex predicate> | <in predicate> | <is null predicate> | <quantified comparison predicate> | <comparison predicate> | <is distinct>)?)
- <exists predicate>
- <xml query>

A boolean predicate or simple expression.

Example:

```
col LIKE 'a%'
```

12.6.86. *comparison operator* ::=

- <eq>
- <ne>
- <ne2>
- <lt>
- <le>
- <gt>
- <ge>

A comparison operator.

Example:

```
=
```

12.6.87. *is distinct* ::=

- IS (NOT)? DISTINCT FROM <common value expression>

Is Distinct Right Hand Side

Example:

```
IS DISTINCT FROM expression
```

12.6.88. *comparison predicate* ::=

- <comparison operator> <common value expression>

A value comparison.

Example:

```
= 'a'
```

12.6.89. *subquery* ::=

- <lparen> (<query expression> | <call statement>) <rparen>

A subquery.

Example:

```
(SELECT * FROM tbl)
```

12.6.90. *quantified comparison predicate* ::=

- <comparison operator> (ANY | SOME | ALL) (<subquery> | (<lparen> <expression> <rparen>))

A subquery comparison.

Example:

```
= ANY (SELECT col FROM tbl)
```

12.6.91. *match predicate* ::=

- (NOT)? (LIKE | (SIMILAR TO)) <common value expression> (ESCAPE <character> | (<lbrace> ESCAPE <character> <rbrace>))?

Matches based upon a pattern.

Example:

```
LIKE 'a_'
```

12.6.92. *like regex predicate* ::=

- (NOT)? LIKE_REGEX <common value expression>

A regular expression match.

Example:

```
LIKE_REGEX 'a.*b'
```

12.6.93. *character* ::=

- <string>

A single character.

Example:

```
'a'
```

12.6.94. *between predicate* ::=

- (NOT)? BETWEEN <common value expression> AND <common value expression>

A comparison between two values.

Example:

```
BETWEEN 1 AND 5
```

12.6.95. *is null predicate* ::=

- IS (NOT)? NULL

A null test.

Example:

```
IS NOT NULL
```

12.6.96. *in predicate* ::=

- (NOT)? IN (<subquery> | (<lparen> <common value expression> (<comma> <common value expression>)* <rparen>))

A comparison with multiple values.

Example:

```
IN (1, 5)
```

12.6.97. *exists predicate* ::=

- EXISTS <subquery>

A test if rows exist.

Example:

```
EXISTS (SELECT col FROM tbl)
```

12.6.98. *group by clause* ::=

- GROUP BY (ROLLUP <lparen> <expression list> <rparen> | <expression list>)

Defines the grouping columns

Example:

```
GROUP BY col1, col2
```

12.6.99. *having clause* ::=

- HAVING <condition>

Search condition applied after grouping.

Example:

```
HAVING max(col1) = 5
```

12.6.100. *order by clause* ::=

- ORDER BY <sort specification> (<comma> <sort specification>)*

Specifices row ordering.

Example:

```
ORDER BY x, y DESC
```

12.6.101. *sort specification* ::=

- <sort key> (ASC | DESC)? (NULLS (FIRST | LAST))?

Defines how to sort on a particular expression

Example:

```
col1 NULLS FIRST
```

12.6.102. *sort key* ::=

- <expression>

A sort expression.

Example:

```
col1
```

12.6.103. *integer parameter* ::=

- <unsigned integer>
- <unsigned value expression primary >

A literal integer or parameter reference to an integer.

Example:

```
?
```

12.6.104. *limit clause* ::=

- (LIMIT <integer parameter> ((<comma> <integer parameter>) | (OFFSET <integer parameter>))?)
- (OFFSET <integer parameter> (ROW | ROWS) (<fetch clause>)?)
- <fetch clause>

Limits and/or offsets the resultant rows.

Example:

```
LIMIT 2
```

12.6.105. *fetch clause* ::=

- FETCH (FIRST | NEXT) (<integer parameter>)? (ROW | ROWS) ONLY

ANSI limit.

Example:

```
FETCH FIRST 1 ROWS ONLY
```

12.6.106. *option clause* ::=

- OPTION (MAKEDEP <identifier> <make dep options > (<comma> <identifier> <make dep options>)* | MAKEIND <identifier> <make dep options > (<comma> <identifier> <make dep options>)* | MAKENOTDEP <identifier> (<comma> <identifier>)* | NOCACHE (<identifier> (<comma> <identifier>)*)?)*

Specifies query options.

Example:

```
OPTION MAKEDEP tbl
```

12.6.107. *expression* ::=

- <condition>

A value.

Example:

```
col1
```

12.6.108. *common value expression* ::=

- (<numeric value expression> ((<double_amp_op> | <concat_op>) <numeric value expression>)*)

Establishes the precedence of concat.

Example:

```
'a' || 'b'
```

12.6.109. *numeric value expression* ::=

- (<term> (<plus or minus> <term>)*)

Example:

```
1 + 2
```

12.6.110. *plus or minus* ::=

- <plus>
- <minus>

The + or - operator.

Example:

```
+
```

12.6.111. *term* ::=

- (<value expression primary> (<star or slash> <value expression primary>)*)

A numeric term

Example:

```
1 * 2
```

12.6.112. *star or slash* ::=

- <star>
- <slash>

The * or / operator.

Example:

```
/
```

12.6.113. *value expression primary* ::=

- <non numeric literal>
- (<plus or minus>)? (<unsigned numeric literal> | (<unsigned value expression primary > (<lsbrace> <numeric value expression> <rsbrace>)*))

A simple value expression.

Example:

```
+col1
```

12.6.114. *parameter reference* ::=

- <qmark>
- (<dollar> <unsigned integer>)

A parameter reference to be bound later.

Example:

```
?
```

12.6.115. *unescapedFunction* ::=

- ((<text aggreate function> | <standard aggregate function> | <ordered aggregate function>) (<filter clause>)? (<window specification>)?) | (<analytic aggregate function> (<filter clause>)? <window specification>) | (<function> (<window specification>)?)
- (XMLCAST <lparen> <expression> AS <data type> <rparen>)

12.6.116. *nested expression* ::=

- (<lparen> (<expression> (<comma> <expression>)*)? (<comma>)? <rparen>)

An expression nested in parens

Example:

```
(1)
```

12.6.117. *unsigned value expression primary* ::=

- <parameter reference>
- (<escaped function> <function> <rbrace>)
- <unescapedFunction>
- <identifier> | <non-reserved identifier>
- <subquery>

- <nested expression>
- <ARRAY expression constructor>
- <searched case expression>
- <case expression>

An unsigned simple value expression.

Example:

```
col1
```

12.6.118. *ARRAY expression constructor* ::=

- ARRAY ((<lsbrace> (<expression> (<comma> <expression>)*)? <rsbrace>) | (<lparen> <query expression> <rparen>))

Creates and array of the given expressions.

Example:

```
ARRAY[1,2]
```

12.6.119. *window specification* ::=

- OVER <lparen> (PARTITION BY <expression list>)? (<order by clause>)? (<window frame>)? <rparen>

The window specification for an analytical or windowed aggregate function.

Example:

```
OVER (PARTION BY col1)
```

12.6.120. *window frame* ::=

- (RANGE | ROWS) ((BETWEEN <window frame bound> AND <window frame bound>) | <window frame bound>)

Defines the mode, start, and optionally end of the window frame

Example:

```
RANGE UNBOUNDED PRECEDING
```

12.6.121. *window frame bound* ::=

- ((UNBOUNDED | <unsigned integer>) (FOLLOWING | PRECEDING))
- (CURRENT ROW)

Defines the start or end of a window frame

Example:

```
CURRENT ROW
```

12.6.122. *case expression* ::=

- CASE <expression> (WHEN <expression> THEN <expression>)+ (ELSE <expression>)? END

If/then/else chain using a common search predicand.

Example:

```
CASE col1 WHEN 'a' THEN 1 ELSE 2
```

12.6.123. *searched case expression* ::=

- CASE (WHEN <condition> THEN <expression>)+ (ELSE <expression>)? END

If/then/else chain using multiple search conditions.

Example:

```
CASE WHEN x = 'a' THEN 1 WHEN y = 'b' THEN 2
```

12.6.124. *function* ::=

- (CONVERT <lparen> <expression> <comma> <data type> <rparen>)
- (CAST <lparen> <expression> AS <data type> <rparen>)
- (SUBSTRING <lparen> <expression> ((FROM <expression> (FOR <expression>)?) | (<comma> <expression list>)) <rparen>)
- (EXTRACT <lparen> (YEAR | MONTH | DAY | HOUR | MINUTE | SECOND | QUARTER | EPOCH) FROM <expression> <rparen>)
- (TRIM <lparen> ((((LEADING | TRAILING | BOTH) (<expression>)?) | <expression>) FROM)? <expression> <rparen>)
- ((TO_CHARS | TO_BYTES) <lparen> <expression> <comma> <string> (<comma> <expression>)? <rparen>)
- ((TIMESTAMPADD | TIMESTAMPDIFF) <lparen> <time interval> <comma> <expression> <comma> <expression> <rparen>)
- <querystring function>
- ((LEFT | RIGHT | CHAR | USER | YEAR | MONTH | HOUR | MINUTE | SECOND | XMLCONCAT | XMLCOMMENT | XMLTEXT) <lparen> (<expression list>)? <rparen>)
- ((TRANSLATE | INSERT) <lparen> (<expression list>)? <rparen>)

- <xml parse>
- <xml element>
- (XMLPI <lparen> ((NAME)? <identifier>) (<comma> <expression>)? <rparen>)
- <xml forest>
- <json object>
- <xml serialize>
- <xml query>
- (POSITION <lparen> <common value expression> IN <common value expression> <rparen>)
- (LISTAGG <lparen> <expression> (<comma> <string>)? <rparen> WITHIN GROUP <lparen> <order by clause> <rparen>)
- (<identifier> <lparen> (ALL | DISTINCT)? (<expression list>)? (<order by clause>)? <rparen> (<filter clause>)?)
- (CURRENT_DATE (<lparen> <rparen>)?)
- ((CURRENT_TIMESTAMP | CURRENT_TIME) (<lparen> <unsigned integer> <rparen>)?)

Calls a scalar function.

Example:

```
func('1', col1)
```

12.6.125. *xml parse* ::=

- XMLPARSE <lparen> (DOCUMENT | CONTENT) <expression> (WELLFORMED)? <rparen>

Parses the given value as XML.

Example:

```
XMLPARSE(DOCUMENT doc WELLFORMED)
```

12.6.126. *querystring function* ::=

- QUERYSTRING <lparen> <expression> (<comma> <derived column>)* <rparen>

Produces a URL query string from the given arguments.

Example:

```
QUERYSTRING('path', col1 AS opt, col2 AS val)
```

12.6.127. *xml element* ::=

- XMLELEMENT <lparen> ((NAME)? <identifier>) (<comma> <xml namespaces>)? (<comma> <xml attributes>)? (<comma> <expression>)* <rparen>

Creates an XML element.

Example:

```
XMLELEMENT(NAME "root", child)
```

12.6.128. *xml attributes* ::=

- XMLATTRIBUTES <lparen> <derived column> (<comma> <derived column>)* <rparen>

Creates attributes for the containing element.

Example:

```
XMLATTRIBUTES(col1 AS attr1, col2 AS attr2)
```

12.6.129. *json object* ::=

- JSONOBJECT <lparen> <derived column list> <rparen>

Produces a JSON object containing name value pairs.

Example:

```
JSONOBJECT(col1 AS val1, col2 AS val2)
```

12.6.130. *derived column list* ::=

- <derived column> (<comma> <derived column>)*

a list of name value pairs

Example:

```
col1 AS val1, col2 AS val2
```

12.6.131. *xml forest* ::=

- XMLFOREST <lparen> (<xml namespaces> <comma>)? <derived column list> <rparen>

Produces an element for each derived column.

Example:

```
XMLFOREST(col1 AS ELEM1, col2 AS ELEM2)
```

12.6.132. *xml namespaces* ::=

- XMLNAMESPACES <lparen> <xml namespace element> (<comma> <xml namespace element>)* <rparen>

Defines XML namespace URI/prefix combinations

Example:

```
XMLNAMESPACES('http://foo' AS foo)
```

12.6.133. *xml namespace element* ::=

- (<string> AS <identifier>)
- (NO DEFAULT)
- (DEFAULT <string>)

An xml namespace

Example:

```
NO DEFAULT
```

12.6.134. *simple data type* ::=

- (STRING (<lparen> <unsigned integer> <rparen>)?)
- (VARCHAR (<lparen> <unsigned integer> <rparen>)?)
- BOOLEAN
- BYTE
- TINYINT
- SHORT
- SMALLINT
- (CHAR (<lparen> <unsigned integer> <rparen>)?)
- INTEGER
- LONG
- BIGINT
- (BIGINTEGER (<lparen> <unsigned integer> <rparen>)?)
- FLOAT
- REAL
- DOUBLE
- (BIGDECIMAL (<lparen> <unsigned integer> (<comma> <unsigned integer>)? <rparen>)?)

- (DECIMAL (<lparen> <unsigned integer> (<comma> <unsigned integer>)? <rparen>)?)
- DATE
- TIME
- (TIMESTAMP (<lparen> <unsigned integer> <rparen>)?)
- (OBJECT (<lparen> <unsigned integer> <rparen>)?)
- (BLOB (<lparen> <unsigned integer> <rparen>)?)
- (CLOB (<lparen> <unsigned integer> <rparen>)?)
- JSON
- (VARBINARY (<lparen> <unsigned integer> <rparen>)?)
- GEOMETRY
- GEOGRAPHY
- XML

A non-collection data type.

Example:

```
STRING
```

12.6.135. *basic data type* ::=

- <simple data type> (<lsbrace> <rsbrace>)*

A data type.

Example:

```
STRING[]
```

12.6.136. *data type* ::=

- <basic data type>
- ((<identifier> | <basicNonReserved>) (<lsbrace> <rsbrace>)*)

A data type.

Example:

```
STRING[]
```

12.6.137. *time interval* ::=

- SQL_TSI_FRAC_SECOND
- SQL_TSI_SECOND
- SQL_TSI_MINUTE
- SQL_TSI_HOUR
- SQL_TSI_DAY
- SQL_TSI_WEEK
- SQL_TSI_MONTH
- SQL_TSI_QUARTER
- SQL_TSI_YEAR

A time interval keyword.

Example:

```
SQL_TSI_HOUR
```

12.6.138. *non numeric literal* ::=

- <string>
- <binary string literal>
- FALSE
- TRUE
- UNKNOWN
- NULL
- (<escaped type> <string> <rbrace>)
- ((DATE | TIME | TIMESTAMP) <string>)

An escaped or simple non numeric literal.

Example:

```
'a'
```

12.6.139. *unsigned numeric literal* ::=

- <unsigned integer literal>
- <approximate numeric literal>
- <decimal numeric literal>

An unsigned numeric literal value.

Example:

```
1.234
```

12.6.140. *ddl statement* ::=

- <create table> (<create table> | <create procedure>)?
- <option namespace>
- <alterStatement>
- <create trigger>
- <create a domain or type alias>
- <create server>
- <create role >
- <drop role >
- <GRANT>
- <Revoke GRANT>
- <CREATE POLICY>
- <DROP POLICY>
- <drop server>
- <drop table>
- <Import foreign schema>
- <Import another Database>
- <create database>
- <use database>
- <drop schema>
- <set schema>
- <create schema>
- <create procedure> (<ddl statement>)?
- <create data wrapper>
- <Drop data wrapper>
- <drop procedure>

A data definition statement.

Example:

```
CREATE FOREIGN TABLE X (Y STRING)
```

12.6.141. *option namespace* ::=

- SET NAMESPACE <string> AS <identifier>

A namespace used to shorten the full name of an option key.

Example:

```
SET NAMESPACE 'http://foo' AS foo
```

12.6.142. *create database* ::=

- CREATE DATABASE <identifier> (VERSION <string>)? (<options clause>)?

create a new database

Example:

```
CREATE DATABASE foo OPTIONS(x 'y')
```

12.6.143. *use database* ::=

- USE DATABASE <identifier> (VERSION <string>)?

database into working context

Example:

```
USE DATABASE foo
```

12.6.144. *create schema* ::=

- CREATE (VIRTUAL | FOREIGN)? SCHEMA <identifier> (SERVER <identifier list>)? (<options clause>)?

create a schema in database

Example:

```
CREATE VIRTUAL SCHEMA foo SERVER (s1,s2,s3);
```

12.6.145. *drop schema* ::=

- DROP (VIRTUAL | FOREIGN)? SCHEMA <identifier>

drop a schema in database

Example:

```
DROP SCHEMA foo
```

12.6.146. *set schema* ::=

- SET SCHEMA <identifier>

set the schema for subsequent ddl statements

Example:

```
SET SCHEMA foo
```

12.6.147. *create a domain or type alias* ::=

- CREATE DOMAIN (<identifier> | <basicNonReserved>) (AS)? <data type> (NOT NULL)?

creates a named type with optional constraints

Example:

```
CREATE DOMAIN my_type AS INTEGER NOT NULL
```

12.6.148. *create data wrapper* ::=

- CREATE FOREIGN ((DATA WRAPPER) | TRANSLATOR) <Unqualified identifier> ((TYPE | HANDLER) <identifier>)? (<options clause>)?

Defines a translator; use the options to override the translator properties.

Example:

```
CREATE FOREIGN DATA WRAPPER wrapper OPTIONS (x true)
```

12.6.149. *Drop data wrapper* ::=

- DROP FOREIGN ((DATA WRAPPER) | TRANSLATOR) <identifier>

Deletes a translator

Example:

```
DROP FOREIGN DATA WRAPPER wrapper
```

12.6.150. *create role* ::=

- CREATE ROLE <Unqualified identifier> (WITH <with role>)?

Defines data role for the database

Example:

```
CREATE ROLE lowly WITH FOREIGN ROLE "role"
```

12.6.151. *with role* ::=

- (ANY AUTHENTICATED)
- ((JAAS | FOREIGN) ROLE <identifier list>)

12.6.152. *drop role* ::=

- DROP ROLE <identifier>

Removes data role for the database

Example:

```
DROP ROLE <data-role>
```

12.6.153. *CREATE POLICY* ::=

- CREATE POLICY <identifier> ON ((<identifier> (FOR (ALL | ((SELECT | INSERT | UPDATE | DELETE) (<comma> (SELECT | INSERT | UPDATE | DELETE))*)))?) | (PROCEDURE <identifier> (FOR ALL)?)) TO <identifier> USING <lparen> <boolean primary> <rparen>

CREATE row level policy

Example:

```
CREATE POLICY pname ON tbl FOR SELECT,INSERT TO role USING col = user();
```

12.6.154. *DROP POLICY* ::=

- DROP POLICY <identifier> ON (<identifier> | (PROCEDURE <identifier>)) TO <identifier>

DROP row level policy

Example:

```
----DROP POLICY pname ON tbl TO role
----
```

12.6.155. *GRANT* ::=

- GRANT (((<grant type> (<comma> <grant type>)*)? ON (TABLE <identifier> (CONDITION ((NOT)? CONSTRAINT)? <string>)? | FUNCTION <identifier> | PROCEDURE <identifier> (CONDITION ((NOT)? CONSTRAINT)? <string>)? | SCHEMA <identifier> | COLUMN <identifier> (MASK (ORDER <unsigned integer>)? <string>)?)) | (ALL PRIVILEGES) | (TEMPORARY TABLE) | (USAGE ON LANGUAGE <identifier>)) TO <identifier>

Defines GRANT for a role

Example:

```
GRANT SELECT ON TABLE x.y TO role
```

12.6.156. *Revoke GRANT* ::=

- REVOKE (((<grant type> (<comma> <grant type>)*)? ON (TABLE <identifier> (CONDITION)? | FUNCTION <identifier> | PROCEDURE <identifier> (CONDITION)? | SCHEMA <identifier> | COLUMN <identifier> (MASK)?)) | (ALL PRIVILEGES) | (TEMPORARY TABLE) | (USAGE ON LANGUAGE <identifier>)) FROM <identifier>

Revokes GRANT for a role

Example:

```
REVOKE SELECT ON TABLE x.y TO role
```

12.6.157. *create server* ::=

- CREATE SERVER <Unqualified identifier> (TYPE <string>)? (VERSION <string>)? FOREIGN ((DATA WRAPPER) | TRANSLATOR) <Unqualified identifier> (<options clause>)?

Defines a connection to a source

Example:

```
CREATE SERVER "h2-connector" FOREIGN DATA WRAPPER h2 OPTIONS ("resource-name"
'java:/accounts-ds');
```

12.6.158. *drop server* ::=

- DROP SERVER <identifier>

Defines dropping connection to foreign source

Example:

```
DROP SERVER server_name
```

12.6.159. *create procedure* ::=

- CREATE (VIRTUAL | FOREIGN)? (PROCEDURE | FUNCTION) <Unqualified identifier> (<lparen> (<procedure parameter> (<comma> <procedure parameter>)*)? <rparen> (RETURNS (<options clause>)? (((TABLE)? <lparen> <procedure result column > (<comma> <procedure result column >)* <rparen>) | <data type>))? (<options clause>)? (AS <statement>)?)

Defines a procedure or function invocation.

Example:

```
CREATE FOREIGN PROCEDURE proc (param STRING) RETURNS STRING
```

12.6.160. *drop procedure* ::=

- DROP (VIRTUAL | FOREIGN)? (PROCEDURE | FUNCTION) <identifier>

Drops a table or view.

Example:

```
DROP FOREIGN TABLE table-name
```

12.6.161. *procedure parameter* ::=

- (IN | OUT | INOUT | VARIADIC)? <identifier> <data type> (NOT NULL)? (RESULT)? (DEFAULT <expression>)? (<options clause>)?

A procedure or function parameter

Example:

```
OUT x INTEGER
```

12.6.162. *procedure result column* ::=

- <identifier> <data type> (NOT NULL)? (<options clause>)?

A procedure result column.

Example:

```
x INTEGER
```

12.6.163. *create table* ::=

- CREATE (<create view> | <create foreign or global temporary table>)

Defines a table or view.

Example:

```
CREATE VIEW vw AS SELECT 1
```

12.6.164. *create foreign or global temporary table* ::=

- ((FOREIGN TABLE) | (GLOBAL TEMPORARY TABLE)) <Unqualified identifier> <create table body>

Defines a foreign or global temporary table.

Example:

```
FOREIGN TABLE ft (col integer)
```

12.6.165. *create view* ::=

- (VIRTUAL)? VIEW <Unqualified identifier> (<create view body > | (<options clause>)?) AS <query expression>

Defines a view.

Example:

```
VIEW vw AS SELECT 1
```

12.6.166. *drop table* ::=

- DROP ((FOREIGN TABLE) | ((VIRTUAL)? VIEW) | (GLOBAL TEMPORARY TABLE)) <identifier>

Drops a table or view.

Example:

```
DROP VIEW name
```

12.6.167. *create foreign temp table* ::=

- CREATE (LOCAL)? FOREIGN TEMPORARY TABLE <Unqualified identifier> <create table body> ON <identifier>

Defines a foreign temp table

Example:

```
CREATE FOREIGN TEMPORARY TABLE t (x string) ON z
```

12.6.168. *create table body* ::=

- <lparen> <table element> (<comma> (<table constraint> | <table element>))* <rparen> (<options clause>)?

Defines a table.

Example:

```
(x string) OPTIONS (CARDINALITY 100)
```

12.6.169. *create view body* ::=

- <lparen> <view element> (<comma> (<table constraint> | <view element>))* <rparen> (<options clause>)?

Defines a view.

Example:

```
(x) OPTIONS (CARDINALITY 100)
```

12.6.170. *table constraint* ::=

- (CONSTRAINT <identifier>)? (<primary key> | <other constraints> | <foreign key>) (<options clause>)?

Defines a constraint on a table or view.

Example:

```
FOREIGN KEY (a, b) REFERENCES tbl (x, y)
```

12.6.171. *foreign key* ::=

- FOREIGN KEY <column list> REFERENCES <identifier> (<column list>)?

Defines the foreign key referential constraint.

Example:

```
FOREIGN KEY (a, b) REFERENCES tbl (x, y)
```

12.6.172. *primary key* ::=

- PRIMARY KEY <column list>

Defines the primary key.

Example:

```
PRIMARY KEY (a, b)
```

12.6.173. *other constraints* ::=

- ((UNIQUE | ACCESSPATTERN) <column list>)
- (INDEX <lparen> <expression list> <rparen>)

Defines ACCESSPATTERN and UNIQUE constraints and INDEXes.

Example:

```
UNIQUE (a)
```

12.6.174. *column list* ::=

- <lparen> <identifier> (<comma> <identifier>)* <rparen>

A list of column names.

Example:

```
(a, b)
```

12.6.175. *table element* ::=

- <identifier> (SERIAL | (<data type> (NOT NULL)? (AUTO_INCREMENT)?)) <post create column>

Defines a table column.

Example:

```
x INTEGER NOT NULL
```

12.6.176. *view element* ::=

- <identifier> (SERIAL | (<data type> (NOT NULL)? (AUTO_INCREMENT)?))? <post create column>

Defines a view column with optional type.

Example:

```
x INTEGER NOT NULL
```

12.6.177. *post create column* ::=

- (<inline constraint>)? (DEFAULT <expression>)? (<options clause>)?

Common options trailing a column

Example:

```
PRIMARY KEY
```

12.6.178. *inline constraint* ::=

- (PRIMARY KEY)
- UNIQUE
- INDEX

Defines a constraint on a single column

Example:

```
x INTEGER PRIMARY KEY
```

12.6.179. *options clause* ::=

- OPTIONS <lparen> <option pair> (<comma> <option pair>)* <rparen>

A list of statement options.

Example:

> OPTIONS ('x' 'y', 'a' 'b')

12.6.180. *option pair* ::=

- <identifier> (<non numeric literal> | (<plus or minus>)? <unsigned numeric literal>)

An option key/value pair.

Example:

> 'key' 'value'

12.6.181. *alter option pair* ::=

- <identifier> (<non numeric literal> | (<plus or minus>)? <unsigned numeric literal>)

Alter An option key/value pair.

Example:

> 'key' 'value'

12.6.182. *alterStatement* ::=

- ALTER (<ALTER TABLE > | <ALTER PROCEDURE > | <ALTER TRIGGER > | <ALTER SERVER > | <ALTER DATA WRAPPER > | <ALTER DATABASE >)

12.6.183. *ALTER TABLE* ::=

- (((VIRTUAL)? VIEW <identifier>) | ((FOREIGN)? TABLE <identifier>)) ((AS <query expression>) | <ADD column> | <ADD constraint> | <alter options list> | <DROP column> | (ALTER COLUMN <alter column options>) | (RENAME (<RENAME Table> | (COLUMN <rename column options>))))

alters options of database

Example:

> ALTER TABLE foo ADD COLUMN x xml

12.6.184. *RENAME Table* ::=

- TO <identifier>

alters table name

Example:

```
ALTER TABLE foo RENAME TO BAR;
```

12.6.185. *ADD constraint* ::=

- ADD <table constraint>

alters table and adds a constraint

Example:

```
ADD PRIMARY KEY (ID)
```

12.6.186. *ADD column* ::=

- ADD COLUMN <table element>

alters table and adds a column

Example:

```
ADD COLUMN bar type OPTIONS (ADD updatable true)
```

12.6.187. *DROP column* ::=

- DROP COLUMN <identifier>

alters table and adds a column

Example:

```
DROP COLUMN bar
```

12.6.188. *alter column options* ::=

- <identifier> ((TYPE (SERIAL | (<data type> (NOT NULL)? (AUTO_INCREMENT)?))) | <alter child options list>)

alters a set of column options

Example:

```
ALTER COLUMN bar OPTIONS (ADD updatable true)
```

12.6.189. *rename column options* ::=

- <identifier> TO <identifier>

renames either a table column or procedure's parameter name

Example:

```
RENAME COLUMN bar TO foo
```

■

12.6.190. *ALTER PROCEDURE* ::=

- (VIRTUAL | FOREIGN)? PROCEDURE <identifier> ((AS <statement>) | <alter options list> | (ALTER PARAMETER <alter column options>) | (RENAME PARAMETER <rename column options>))

alters options of database

Example:

```
ALTER PROCEDURE foo OPTIONS (ADD x y)
```

12.6.191. *ALTER TRIGGER* ::=

- TRIGGER ON <identifier> INSTEAD OF (INSERT | UPDATE | DELETE) (AS <for each row trigger action> | ENABLED | DISABLED)

alters options of table triggers

Example:

```
ALTER TRIGGER ON vw INSTEAD OF INSERT ENABLED
```

12.6.192. *ALTER SERVER* ::=

- SERVER <identifier> <alter options list>

alters options of database

Example:

```
ALTER SERVER foo OPTIONS (ADD x y)
```

12.6.193. *ALTER DATA WRAPPER* ::=

- ((DATA WRAPPER) | TRANSLATOR) <identifier> <alter options list>

alters options of data wrapper

Example:

```
ALTER DATA WRAPPER foo OPTIONS (ADD x y)
```

12.6.194. *ALTER DATABASE* ::=

- DATABASE <identifier> <alter options list>

alters options of database

Example:

```
ALTER DATABASE foo OPTIONS (ADD x y)
```

12.6.195. *alter options list* ::=

- OPTIONS <lparen> (<add set option > | <drop option>) (<comma> (<add set option > | <drop option>))* <rparen>

a list of alterations to options

Example:

```
OPTIONS (ADD updatable true)
```

12.6.196. *drop option* ::=

- DROP <identifier>

drop option

Example:

```
DROP updatable
```

12.6.197. *add set option* ::=

- (ADD | SET) <alter option pair>

add or set an option pair

Example:

```
ADD updatable true
```

12.6.198. *alter child options list* ::=

- OPTIONS <lparen> (<add set child option > | <drop option>) (<comma> (<add set child option > | <drop option>))* <rparen>

a list of alterations to options

Example:

```
OPTIONS (ADD updatable true)
```

12.6.199. *drop option* ::=

- DROP <identifier>

drop option

Example:

> DROP updatable

12.6.200. *add set child option* ::=

- (ADD | SET) <alter child option pair>

add or set an option pair

Example:

> ADD updatable true

12.6.201. *alter child option pair* ::=

- <identifier> (<non numeric literal> | (<plus or minus>)? <unsigned numeric literal>)

Alter An option key/value pair.

Example:

> 'key' 'value'

12.6.202. *Import foreign schema* ::=

- IMPORT (FOREIGN SCHEMA <identifier>)? FROM (SERVER | REPOSITORY) <identifier> INTO <identifier> (<options clause>)?

imports schema metadata from server

Example:

> IMPORT FOREIGN SCHEMA foo FROM SERVER bar

12.6.203. *Import another Database* ::=

- IMPORT DATABASE <identifier> VERSION <string> (WITH ACCESS CONTROL)?

imports another database into current database

Example:

> IMPORT DATABASE vdb VERSION '1.2.3' WITH ACCESS CONTROL]

12.6.204. *identifier list* ::=

- <identifier> (<comma> <identifier>)*

12.6.205. *grant type* ::=

- SELECT

- INSERT
- UPDATE
- DELETE
- EXECUTE
- ALTER
- DROP