Table of Contents

Introduction 1.1

Legal Notice 1.2

Reference Guide 2.1

- Data Sources 2.1.1
- Virtual Databases 2.1.2
  - Developing a Virtual Database 2.1.2.1
  - DDL VDB 2.1.2.2
  - Using XML & DDL 2.1.2.3
  - VDB Properties 2.1.2.4
  - Schema Object DDL 2.1.2.5
  - Domain DDL 2.1.2.6
  - MultiSource Models 2.1.2.7
  - Metadata Repositories 2.1.2.8
  - REST Service Through VDB 2.1.2.9
  - VDB Reuse 2.1.2.10
- SQL Support 2.1.3
  - Identifiers 2.1.3.1
  - Expressions 2.1.3.2
  - Criteria 2.1.3.3
  - Scalar Functions 2.1.3.4
    - Numeric Functions 2.1.3.4.1
    - String Functions 2.1.3.4.2
    - Date_Time Functions 2.1.3.4.3
    - Type Conversion Functions 2.1.3.4.4
    - Choice Functions 2.1.3.4.5
    - Decode Functions 2.1.3.4.6
    - Lookup Function 2.1.3.4.7
    - System Functions 2.1.3.4.8
    - XML Functions 2.1.3.4.9
    - JSON Functions 2.1.3.4.10
    - Security Functions 2.1.3.4.11
    - Spatial Functions 2.1.3.4.12
    - Miscellaneous Functions 2.1.3.4.13
    - Nondeterministic Function Handling 2.1.3.4.14
  - DML Commands 2.1.3.5
    - Set Operations 2.1.3.5.1
    - Subqueries 2.1.3.5.2

WITH Clause 2.1.3.5.3
SELECT Clause 2.1.3.5.4
FROM Clause 2.1.3.5.5
XMLTABLE 2.1.3.5.5.1
ARRAYTABLE 2.1.3.5.5.2
OBJECTTABLE 2.1.3.5.5.3
TEXTTABLE 2.1.3.5.5.4
WHERE Clause 2.1.3.5.6
GROUP BY Clause 2.1.3.5.7
HAVING Clause 2.1.3.5.8
ORDER BY Clause 2.1.3.5.9
LIMIT Clause 2.1.3.5.10
INTO Clause 2.1.3.5.11
OPTION Clause 2.1.3.5.12
DDL Commands 2.1.3.6
Temp Tables 2.1.3.6.1
Alter View 2.1.3.6.2
Alter Procedure 2.1.3.6.3
Alter Trigger 2.1.3.6.4
Procedures 2.1.3.7
Procedure Language 2.1.3.7.1
Virtual Procedures 2.1.3.7.2
Update Procedures 2.1.3.7.3
Comments 2.1.3.8
Datatypes 2.1.4
Supported Types 2.1.4.1
Type Conversions 2.1.4.2
Special Conversion Cases 2.1.4.3
Escaped Literal Syntax 2.1.4.4
Updatable Views 2.1.5
preserved Table 2.1.5.1
Transaction Support 2.1.6
AutoCommitTxn Execution Property 2.1.6.1
Updating Model Count 2.1.6.2
JDBC and Transactions 2.1.6.3
Transactional Behavior with JBoss Data Source Types 2.1.6.4
Limitations and Workarounds 2.1.6.5
Data Roles 2.1.7
Permissions 2.1.7.1
Role Mapping 2.1.7.2

XML Definition 2.1.7.3
Customizing 2.1.7.4
System Schema 2.1.8
SYS 2.1.8.1
SYSADMIN 2.1.8.2
Translators 2.1.9
Amazon S3 Translator 2.1.9.1
Amazon SimpleDB Translator 2.1.9.2
Apache Accumulo Translator 2.1.9.3
Apache SOLR Translator 2.1.9.4
Cassandra Translator 2.1.9.5
Couchbase Translator 2.1.9.6
Delegating Translators 2.1.9.7
File Translator 2.1.9.8
Google Spreadsheet Translator 2.1.9.9
Infinispan Translator 2.1.9.10
JDBC Translators 2.1.9.11
Actian Vector Translator 2.1.9.11.1
Apache Phoenix Translator 2.1.9.11.2
Cloudera Impala Translator 2.1.9.11.3
DB2 Translator 2.1.9.11.4
Derby Translator 2.1.9.11.5
Exasol Translator 2.1.9.11.6
Greenplum Translator 2.1.9.11.7
H2 Translator 2.1.9.11.8
Hive Translator 2.1.9.11.9
HSQL Translator 2.1.9.11.10
Informix Translator 2.1.9.11.11
Ingres Translators 2.1.9.11.12
Intersystems Cache Translator 2.1.9.11.13
JDBC ANSI Translator 2.1.9.11.14
JDBC Simple Translator 2.1.9.11.15
MetaMatrix Translator 2.1.9.11.16
Microsoft Access Translators 2.1.9.11.17
Microsoft SQL Server Translator 2.1.9.11.18
ModeShape Translator 2.1.9.11.19
MySQL Translators 2.1.9.11.20
Netezza Translator 2.1.9.11.21
Oracle Translator 2.1.9.11.22
OSISoft PI Translator 2.1.9.11.23

PostgreSQL Translator 2.1.9.11.24
PrestoDB Translator 2.1.9.11.25
Redshift Translator 2.1.9.11.26
SAP Hana Translator 2.1.9.11.27
SAP IQ Translator 2.1.9.11.28
Sybase Translator 2.1.9.11.29
Teiid Translator 2.1.9.11.30
Teradata Translator 2.1.9.11.31
Vertica Translator 2.1.9.11.32
JPA Translator 2.1.9.12
LDAP Translator 2.1.9.13
Loopback Translator 2.1.9.14
Microsoft Excel Translator 2.1.9.15
MongoDB Translator 2.1.9.16
OData Translator 2.1.9.17
OData V4 Translator 2.1.9.18
Swagger Translator 2.1.9.19
OLAP Translator 2.1.9.20
Salesforce Translators 2.1.9.21
SAP Gateway Translator 2.1.9.22
Web Services Translator 2.1.9.23
Federated Planning 2.1.10
Planning Overview 2.1.10.1
Query Planner 2.1.10.2
Query Plans 2.1.10.3
Federated Optimizations 2.1.10.4
Subquery Optimization 2.1.10.5
XQuery Optimization 2.1.10.6
Federated Failure Modes 2.1.10.7
Conformed Tables 2.1.10.8
Architecture 2.1.11
Terminology 2.1.11.1
Data Management 2.1.11.2
Query Termination 2.1.11.3
Processing 2.1.11.4
BNF for SQL Grammar 2.1.12

Legal Notice

12.1 Teiid Documentation

Contribute

The documentation project is hosted on GitHub at (teiid/teiid-documents).

It is published on GitHub Pages at (teiid.github.io/teiid-documents/master/content) ('master' can be substituted with any maintained branch e.g. '10.3.x').

For simple changes you can just use the online editing capabilities of GitHub by navigating to the appropriate source file and selecting fork/edit.

For larger changes follow these 3 steps:

Step.1 clone the sources

```
git clone git@github.com:teiid/teiid-documents.git
```

Step.2 do edit

Use any text editor to edit the adoc files, AsciiDoc Syntax Quick Reference can help you in AsciiDoc Syntax.

Step.3 submit your change

Once the pull request is committed the published content will be updated automatically.

Test locally

You may need test locally, to make sure the changes are correct, to do this install gitbook, then execute the following commands from the checkout location:

```
$ gitbook install
$ gitbook serve -w
```

Once above commands execute successfully (may take a few minutes), you should see the "Serving book at …" message and the http format document can be tested locally via `http://localhost:4000/` .

Generate html/pdf/epub/mobi

You may locally create rendered forms of the documentation. To do this install gitbook and ebook-convert, then execute the following commands from the checkout location:

```
$ gitbook build ./ teiid-documents
$ gitbook pdf ./ teiid-documents.pdf
$ gitbook epub ./ teiid-documents.epub
$ gitbook mobi ./ teiid-documents.mobi
```

Once above commands executes successfully, the `teiid-documents` folder, `teiid-documents.pdf` , `teiid-documents.epub` , and `teiid-documents.mobi` will be generated.

CI Build

The .travis.yaml file allows for continuous integration of doc changes on multiple branches to be published to a single gh-pages branch. When you setup the travis build job you must create the gh-pages branch if it does not already exist:

```
git checkout --orphan gh-pages
git rm -rf .
git commit --allow-empty -m "initializing gh-pages"
git push origin gh-pages
```

You will need to add an appropriate user and git api key with repo access as the environment properties GITHUB_USER and GITHUB_API_KEY respectively in the travis build settings.

Legal Notice

1801 Varsity Drive Raleigh, NC27606-2072USA Phone: +1 919 754 3700 Phone: 888 733 4281 Fax: +1 919 754 3701 PO Box 13588 Research Triangle Park, NC27709USA

The GPG fingerprint of the security@redhat.com key is:

CA 20 86 86 2B D6 9D FC 65 F6 EC C4 21 91 80 CD DB 42 A6 0E

Reference Guide

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

Commercial development support, production support, and training for Teiid is available through JBoss Inc. Teiid is a Professional Open Source project and a critical component of the JBoss Enterprise Data Services Platform.

Before one can delve into Teiid it is very important to learn few basic constructs of Teiid, like what is VDB? what is Model? etc. For that please read the short introduction.

Data Sources

Teiid provides the means (i.e., Translators and JEE connectors) to access a variety of types of data sources.

The types of data sources that are currently accessible are:

- Databases
- Web Services
- OData
- Big Data/No SQL/Search Engines/JCR and Other Sources
- Enterprise Systems
- Object Sources
- LDAP
- Files
- Spreadsheets

Databases

See JDBC Translators for access to:

- Oracle
- PostgreSQL
- MySQL/MariaDB
- DB2
- Microsoft SQL Server
- Sybase
- SAP IQ
- Microsoft Access
- Derby
- H2
- HSQL
- Ingres
- Informix
- MetaMatrix
- Teradata
- Vertica
- Exasol
- Generic ANSI SQL - for typical JDBC/ODBC sources
- Simple SQL - for any JDBC/ODBC source

Web Services

See Web Services Translator for access to:

- SOAP
- REST
- Arbitrary HTTP(S)

OData

See the OData Translator

Big Data/No SQL/Search Engines/JCR and Other Sources

- Actian Vector
- Amazon S3
- Amazon SimpleDB
- Apache Accumulo
- Apache Cassandra DB
- Apache SOLR
- Apache Spark
- Couchebase
- Greenplum
- Hive / Hadoop / Amazon Elastic MapReduce
- Impala / Hadoop / Amazon Elastic MapReduce
- ModeShape JCR Repository
- Mongo DB
- Mondrian OLAP
- Netezza data warehouse appliance
- Phoenix / HBase
- PrestoDB
- Redshift

Enterprise Systems

- OSISoft PI
- SalesForce
- SAP Gateway

- SAP Hana
- Teiid

Object Sources

- Infinispan HotRod Mode
- Intersystems Cache Object Database
- JPA sources

LDAP

See the LDAP Translator for access to:

- RedHat Directory Server
- Active Directory

Files

See the File Translator for use with:

- Delimited/Fixed width
- XML

Spreadsheets

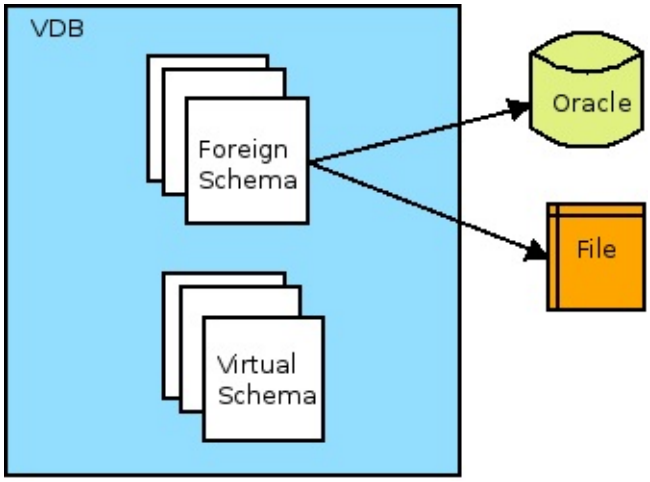
- Excel
- Google Spreadsheet

This represents data sources that have been validated to work using the available translators and connectors. However, this does not preclude a new data source from working. It can be as easy as extending an existing translator, to creating a new translator using the Translator Development extensions.

Take a look at the list of Translators that are used as the bridge between Teiid and the external system.

Virtual Databases

A virtual database (or 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 VDB typically contains multiple schema components (also called as models), and each schema contains the metadata (tables, procedures, functions). There are two (2) different types of schemas

- Source Schema (also called Physical or Foreign schema), which represents an external/remote data sources like Relational database (Oracle, DB2, MySQL..), Files(CSV, Excel..), Web-Services(SOAP, REST) etc.
- Virtual Schema. This is a view layer or logical schema layer, that is defined using schema objects from Foreign Schemas. For example, creating a view table using multiple foreign tables from different sources, thus hiding the complexities of definition of the view from user.

One important thing to note is, a VDB ONLY contains metadata, NEVER copies/has the actual data. Any usecase involving Teiid MUST have a VDB to begin with. So, it is very important to learn how a VDB can be designed/developed.

Below is an example VDB, that is using a single foreign schema component defining a connection to PostgreSQL database.

Example: 1

```
<vdb name="my-example" version="1">
    <model name="test" type="PHYSICAL">
        <property name="importer.schemaName" value="public"/>
        <property name="importer.useFullSchemaName" value="false"/>
        <property name="importer.tableTypes" value="TABLE,VIEW"/>
        <source name="pqsql" translator-name="postgresql" connection-jndi-name="java:/postgres-ds"/>
    </model>
</vdb>
```

Another variation of the VDB using completely DDL and using SQL-MED specification.

Example: 2

```
CREATE DATABASE my_example VERSION '1.0.0';
USE DATABASE my_example VERSION '1.0.0';
CREATE FOREIGN DATA WRAPPER postgresql;
CREATE SERVER pgsql TYPE 'postgresql-9.4-1201.jdbc41.jar'
    VERSION 'one' FOREIGN DATA WRAPPER postgresql
    OPTIONS (
        "jndi-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'
);
```

Both formats define the same VDB.

There is lot to be explained from above examples, in the following sections, we will go into detail about each of those lines. Before that we need to learn about further fractions in the *Source Schema* component.

External Data Sources

A "source schema" component in VDB as shown in above example is a collection schema objects as tables, procedures and functions that represent an external data source's metadata locally. In the above example, it did not define any such schema objects directly, but will instead import them from the server. Details of the connection to the external data source were provided through "jndi-name", which is a named connection reference to a external data source.

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

Resource Adapter

A resource adapter (also called as SERVER) is connection object to the external data source. In the case of relational database this can be achieved through a JDBC connection, or in the case of a File this may be a reference to file's location. The resource-adapter provides a unified interface to define a connection in the Teiid. A resource adapter also provides way to natively issue commands and gather results. Teiid provides variety of resource adaptors to many different systems or one can be developed for new/custom data source. A resource adapters connection is represented above as the "jndi-name".

As VDB developer you need to know, how to configure these sources in the Teiid. In WildFly Server these are defined as JCA components. In Teiid embedded, the developer has to define the connections to these sources programmatically. Check out Administrator's Guide on how to configure these in WildFly, or embedded examples, if you are working with Teiid Embedded.

Translator

A Translator (also called DATA WRAPPER) is a component that provides an abstraction layer between Teiid Query Engine and physical data source, that knows how to convert Teiid issued query commands into source specific commands and execute them using the Resource Adapter. It also have smarts to convert the result data that came from the physical source into a form that Teiid Query engine is expecting. For example, when working with a web-service translator, a SQL procedure executed at Teiid layer may be converted to a HTTP based call through a translator, and response JSON could be converted to tabular results.

Teiid provides various translators as part of the system, or one can be developed using the provided java libraries. For list of available Translators see Translators

Important	In a VDB, a source schema **must be** configured with a **correct** Translator and a **valid** resource adapter, to make the system work.

Developing a Virtual Database

There are few different ways a Virtual Database can be developed. Each method has advantages and disadvantages.

A VDB is developed as file artifact, which can deployed into a Teiid Server. This file artifact contains the metadata about the VDB, or contains the details to fetch the metadata from source data sources. These artifacts can be shared and moved between different servers.

- vdb.xml : In this file format, you can use combination of XML elements and DDL elements to define the metadata.
- vdb.ddl : In this file format, you can use strictly DDL using SQL-MED (with few custom extensions) to define the metadata. This can be viewed as next version to the vdb.xml.
- myvdb.vdb : This is an archive based (zip) file format is combination of above vdb.xml or vdb.ddl file enclosed in zip archive along with any other supporting files like externalized DDL files, UDF libraries. This closely resembles the legacy Designer VDB format, however this will not contain any .INDEX or .XMI files. If the individual schema elements inside a given model/schema is large and managability of that schema in a single vdb file is getting hard as with above formats, then consider using this format. With this you can define each model/schema's DDL in its own file. The ZIP archive structure must resemble

```
myvdb.vdb
/META-INF
 vdb.ddl
/schema1.ddl
/schema2.ddl
/lib
  myudf.jar
```

vdb.xml and vdb.ddl may be deployed as standalone files. As a standalone file, the VDB file name pattern must adhere to "-vdb.xxx" for the Teiid VDB deployer to recognize this file.

They may also be contained in a .vdb zip file along with other relevant files, such as jars, additional ddl, and static file resources.

Important	It is important to note that, the metadata represented by the VDB formats is **EXACTLY** same in all different ways. In fact, you can convert a VDB from one type to the other.

Steps to follow in developing a VDB

This will walk through developing a DDL based VDB.

Step 1: Pick Name and Version

Pick the name and version of the virtual database you want to create. From previous example this represents

```
CREATE DATABASE my_example VERSION '1.0.0';
USE DATABASE my_example VERSION '1.0.0';
```

Step 2: Configuring a Source(s)

When working with external sources, there are few extra steps need to be followed, as not all the software components required for the connection nor configuration are automatically provided by Teiid.

Step 2A: Find the Translator

- First find out if the support for the source is provided in Teiid. Look at Teiid documentation and supported translators. Pick the names of translator(s) you will be using. From previous example this represents

```
CREATE FOREIGN DATA WRAPPER postgresql;
```

here "postgresql" is our translator name, as example assumes we are going to query a PostgreSQL database.

Step 2B: Find the module to connect to External Source

- Typically all relational databases are connected using their JDBC drivers. Find out if the external source has a JDBC driver? if this source has JDBC driver, then acquire the driver jar file.
- Once the driver is acquired, then make sure this driver is **Type 4** driver, and then deploy this driver into Teiid server using either web-console application or CLI admin-console. The below example shows deploying the Oracle driver in Teiid Server based on WildFly using CLI admin-console. If driver is **not** Type 4, it can be still used, but more set up is needed.

```
</wildfly/bin>$./jboss-cli.sh --connect
[standalone@localhost:9990 /] deploy /path/to/ojdbc6.jar
```

- if the source does not have JDBC driver and has resource-adapter provided by Teiid, then driver for it is already available in Teiid server. No further action required for this.

Step 2C: Create a Connection to External Source

- Based on above driver or resource adapter a connection to the external source need to be created. There are many methods to create a data source connection.
- Teiid Server (choose one method from below)
  - Edit the *wildfly/standalone/configuration/standalone-teiid.xml* file and add respective data source or resource adapter configuration. The examples of these templates are provided in *wildfly/docs/teiid/datasources* directory.
  - Use Teiid Web-console and follow the directions to create a data source or resource-adapter.
  - Use CLI admin-console and execute the script. The sample scripts are given in *wildfly/docs/teiid/datasources* directory. Also, checkout documentation at Administrator's Guide for more details.
- Teiid Embedded
  - Create the connection programmatically, by supplying your own libraries to connect to the source.

From previous example this represents

```
CREATE SERVER pgsql TYPE 'postgresql-9.4-1201.jdbc41.jar'
    VERSION 'one' FOREIGN DATA WRAPPER postgresql
    OPTIONS (
        "jndi-name" 'java:/postgres-ds'
    );
```

Warning	This probably is most challenging step in terms of understanding Teiid, make sure you follow before going further into next steps.

Step 3: Create Source Schema

Now that access the external sources is defined, "source schema" or models as shown before needs to be created and metadata needs to be defined.

From previous example this represents

```
CREATE SCHEMA test SERVER pgsql;
SET SCHEMA test;
```

SET SCHEMA statement sets the context in which following DDL statements to fall in.

Schema component is defined, but it has no metadata. i.e tables, procedures or functions. These can be defined one of two ways for a source model, either importing the metadata directly from the source system itself, or defining the DDL manually inline in this file.

Step 3A: Import Metadata

- Using the data source connections created in Step 2, import the metadata upon deployment of the VDB. Note that this capability is slightly different for each source, as to what and how/what kind of metadata is. Check individual source's translator documentation for more information. From previous example this represents

```
IMPORT FOREIGN SCHEMA public FROM SERVER pgsql INTO test
    OPTIONS(
        importer.useFullSchemaName false,
        importer.tableTypes 'TABLE,VIEW'
);
```

The above import statement is saying that, import the "public" schema from external data source defined by "pgsql" into local "test" schema in Teiid. It also further configures to only fetch TABLE, VIEW types, and do not use fully qualified schema names in the imported metadata. Each translator/source has many of these configuration options you can use to filter/refine your selections, for more information consult the translator documents at Translators for every source you are trying to connect to.

Step 3B: Define Metadata using DDL

Instead of importing the metadata, you can manually define the tables and procedures inline to define the metadata. This will be further explained in next sections detail on every DDL statement supported. For example, you can define a table like

```
CREATE FOREIGN TABLE CUSTOMER (
    SSN char(10) PRIMARY KEY,
    FIRSTNAME string(64),
    LASTNAME string(64),
    ST_ADDRESS string(256),
    APT_NUMBER string(32),
    CITY string(64),
    STATE string(32),
    ZIPCODE string(10)
);
```

Warning	Please note that when metadata is defined in this manner, the source system must also have representative schema to support any queries resulting from this metadata. Teiid CAN NOT automatically create this structure in your data source. For example, with above table definition, if you are connecting Oracle database, the Oracle database must have the existing table with matching names. Teiid can not create this table in Oracle for you.

- Repeat this Step 2 & Step 3, for all the external data sources to be included in this VDB

Step 5: Create Virtual Views

- Now using the above source's metadata, define the abstract/logical metadata layer using Teiid's DDL syntax. i.e. create VIEWS, PROCEDURES etc to meet the needs of your business layer. For example (pseudo code):

```
CREATE VIRTUAL SCHEMA reports;

CREATE VIEW SalesByRegion (
   quarter date,
   amount decimal,
   region varchar(50)
) AS
  SELECT ... FROM Sales JOIN Region on x = y WHERE ...
```

- Repeat this step as needed any number of Virtual Views you need. You can refer to View tables in one view from others.

Step 6: Deploy the VDB

- Once the VDB is completed, then this VDB needs to be deployed to the Teiid Server. (this is exactly same as you deploying a WAR file for example). One can use Teiid web-console or CLI admin-console to do this job. For example below cli can be used

```
deploy my-vdb.ddl
```

Step 7: Client Access

- Once the VDB is available on the Teiid Server in ACTIVE status, this VDB can be accessed from any JDBC/ODBC connection based applications. You can use BI tools such as Tableau, Business Objects, QuickView, Pentaho by creating a connection to this VDB. You can also access the VDB using OData V4 protocol without any further coding.

No matter how you are developing the VDB, whether you are using the tooling or not, the above are steps to be followed to build a successful VDB.

vdb.xml

The vdb-deployer.xsd schema for this xml file format is available in the schema folder under the docs with the Teiid distribution.

See also link:xml_deployment_mode.adoc

VDB Zip Deployment

For more complicated scenarios you are not limited to just an xml/ddl file deployment. In a vdb zip deployment:

- The deployment must end with the extension .vdb
- The vdb xml file must be zip under /META-INF/vdb.xml
- If a /lib folder exists any jars found underneath will automatically be added to the vdb classpath.
- Files within the VDB zip are accessible by a Custom Metadata Repository using the `MetadataFactory.getVDBResources()` method, which returns a map of all `VDBResources` in the VDB keyed by absolute path relative to the vdb root. The resources are also available at runtime via the SYSADMIN.VDBResources table.
- The built-in DDL-FILE metadata repository type may be used to define DDL-based metadata in other files within the zip archive. This improves the memory footprint of the vdb metadata and the maintainability of the metadata.

Example VDB Zip Structure

```
/META-INF
   vdb.xml
/ddl
   schema1.ddl
/lib
   some-udf.jar
```

In the above example a vdb.xml could use a DDL-FILE metadata type for schema1:

```
<model name="schema1" ...
   <metadata type="DDL-FILE">/ddl/schema1.ddl</metadata>
</model>
```

The contents inside schema1.ddl can include DDL for Schema Objects

DDL VDB

A Virtual Database (VDB) can created through DDL statements. Teiid supports SQL-MED specification to configure the foreign data sources.

A DDL file captures information about the VDB, the sources it integrate, and preferences for importing metadata. The format of the DDL file can be any elements in documented here. The DDL file may be deployed as a single file, or in a zip archive. See Developing a Virtual Database for a discussion of the .vdb zip packaging.

Table of Contents

- DDL File Deployment
- DDL File Format
- Create a Database
- Create a Translator
- Create a Connection To an External Source
- Create SCHEMA in VDB
- Importing Schema
  - Importing another Virtual Database (VDB Reuse)
- Create Schema Objects
- Data Roles
- Differences with vdb.xml metadata

DDL File Deployment

You can simply create a **SOME-NAME-vdb.ddl** file.

Important	The VDB name pattern must adhere to "-vdb.ddl" for the Teiid VDB deployer to recognize this file when deployed in Teiid Server.

Example VDB DDL Template

```
CREATE DATABASE my_example VERSION '1.0.0';
USE DATABASE my_example VERSION '1.0.0';

CREATE FOREIGN DATA WRAPPER postgresql;
CREATE SERVER pgsql TYPE 'postgresql-9.4-1201.jdbc41.jar'
    VERSION 'one' FOREIGN DATA WRAPPER postgresql
    OPTIONS (
        "jndi-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'
);
```

DDL File Format

For compatibility with the existing metadata system, DDL statements must appear in a specific order to define a virtual database. All of the database structure must be defined first - this includes create/alter/drop database, domains, vdb import, roles, and schemas statements. Then the schema object, schema import, and permission DDL may appear.

Create a Database

Every VDB file must start with database definition where it specifies the name and version of the database. The create syntax for database is

```
CREATE DATABASE {db-name} [VERSION {version-string}] OPTIONS ( <options-clause>)

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

An example statement

```
CREATE DATABASE my_example VERSION '1' OPTIONS ("cache-metadata" true);
```

For list database scoped properties see VDB properties.

Immediately following the create database statement is an analogous use database statement.

As we learned about the VDB components earlier in the guide, we need to first create translators, then connections to data sources, and then using these we can gather metadata about these sources. There is no limit on how many translators, or data sources or schemas you create to build VDB.

Create a Translator

A translator is an adapter to the foreign data source. The creation of translator in the context of the VDB creates a reference to the software module that is available in the Teiid system. Some of the examples of available translator modules include:

- oracle
- mysql
- postgresql
- mongodb

See Data Sources for more.

```
CREATE FOREIGN ( DATA WRAPPER | TRANSLATOR ) {translator-name}
    [ TYPE {base-translator-type} ]
    OPTIONS ( <options-clause>)

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

Optional *TYPE* is used to create "override" translator. The *OPTIONS* clause is used to provide the "execution-properties" of a specific translator defined in either in {translator-name} or {base-translator-name}. These names **MUST** match with available Translators in the system. link:Translators.adoc[Translators} documents all the available translators.

Example 1: Example creating translator

```
CREATE FOREIGN DATA WRAPPER postgresql;
```

For all available translators see Translators

1. Example 2: Example creating Override Translator

```
CREATE FOREIGN DATA WRAPPER oracle-override TYPE oracle OPTIONS (useBindVariables
false);
```

The above example creates a translator override with an example showing turning off the prepared statements.

Additional management support to alter, delete a translator

```
ALTER  (DATA WRAPPER|TRANSLATOR) {translator-name} OPTIONS (ADD|SET|DROP <key-
value>);

DROP FOREIGN [<DATA> <WRAPPER>|<TRANSLATOR>] {translator-name}
```

Create a Connection To an External Source

Before you can create a connection to the data source, you must either have a JDBC driver (Type 4) that can connect to the data source, or Teiid system must have provided a resource adapter (RAR) file to enable connection to the data source. If you are using the JDBC driver file this should have already been deployed to the Teiid system, or made it available on the classpath in the case of the Teiid Embedded. There is currently no DDL mechanism to deploy the external drivers.

Now to create connection to the external data source. One needs to know the name of deployment. For JDBC drivers, it is typically JAR name with out path. For resource adapters, it is the name of the resource-adapter. Step also associates the connection created with the translator to be used in communicating with this source.

```
CREATE SERVER {source-name} TYPE '{source-type}'
    [VERSION '{version}'] FOREIGN DATA WRAPPER {translator-name}
    OPTIONS (<options-clause>)

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

Name	Description
source-name	Name given to the source’s connection.
source-type	For JDBC connection, the driver name or resource-adapter name.
translator-name	Name of the translator to be used with this server.
options	All connection properties for the connection.

For all available translators see Translators

Example 3: creating a data source connection to Postgres database

```
CREATE SERVER pgsql TYPE 'postgresql-9.4-1201.jdbc41.jar'
    FOREIGN DATA WRAPPER postgresql
    OPTIONS (
        "jndi-name" 'java:/postgres-ds'
    );
```

The below are the typical properties that need to be configured for a JDBC connection

Name	Description
jndi-name	Jndi name of the datasource

Note	Any additional properties to create a data-source in WildFly can also used here in OPTIONS clause.

Important	If the data source is already exists in the configuration, then supply only provide *jndi-name* property (you can omit all other properties), then above command will create a new connection, but will use existing configuration in the system.

The below shows an example connection with resource adapter.

Example 4: creating a data source connection to "file" resource adapter.

```
CREATE SERVER marketdata TYPE 'file'
    FOREIGN DATA WRAPPER file
    OPTIONS(
        ParentDirectory '/path/to/marketdata'
    );
```

For all available data sources see data sources

Additional management support to alter/delete a connection.

```
ALTER  SERVER {source-name}  OPTIONS ( ADD|SET|DROP <key-value>);
DROP SERVER  {source-name};
```

Warning	ALTER can be used to change properties, but due to a bug in WildFly this feature currently does not work.

Now that we have the Translators and Connections created, the next step is to create SCHEMAs and work with metadata.

Create SCHEMA in VDB

Before metadata about data sources or abstraction layers can be created, a container for this metadata needs to be created. In relational database concepts this is called Schema, and this also works as a namespace in which metadata objects like TABLES, VIEWS and PROCEDURES exist. The below DDL shows how to create a SCHEMA element.

```
CREATE [VIRTUAL] SCHEMA {schema-name}
    [SERVER {server-name} (<COMMA> {server-name})*]
    OPTIONS (<options-clause>)

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

- The use of VIRTUAL keyword defines if this schema is "Virtual Schema". In the absence of the VIRTUAL keyword, this Schema element represents a "Source Schema". Refer to VDB Guide about different types of Schema types.

Important	If the Schema is defined as "Source Schema", then SERVER configuration **must** be provided, to be able to determine the data source connection to be used when executing queries that belong to this Schema. Providing multiple Server names configure this Schema as "multi-source" model. See Multisource Models

	for more information.

The below are the typical properties that need to be configured for a Schema in the OPTIONS clause.

Name	Description
VISIBILITY	Is Schema visible during metadata interrogation

Example 5: Showing to create a source schema for PostgreSQL server from example above

```
CREATE SCHEMA test SERVER pgsql;
```

Additional management support to alter/delete a schema can be done through following commands.

```
ALTER [VIRTUAL] SCHEMA {schema-name} OPTIONS (ADD|SET|DROP <key-value>);
DROP SCHEMA {schema-name};
```

Importing Schema

If you are designing a source schema, you can add the TABLES, PROCEDURES manually to represent the data source, however in certain situations this can be tedious, or complicated. For example, if you need to represent 100s of existing tables from your Oracle database in Teiid? Or if you are working with MongoDB, how are you going to map a document structure into a TABLE? For this purpose, Teiid provides an import metadata command, that can import/create metadata that represents the source. The following command can be used for that purpose with most of the sources (LDAP source is only exception, not providing import)

```
IMPORT FOREIGN SCHEMA {foreign-schema-name}
    FROM (SERVER {server-name} | REPOSITORY {repository-name})
    INTO {schema-name}
    OPTIONS (<options-clause>)

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

foreign-schema-name : Name of schema in external data source to import. Typically most databases are tied to a schema name, like "public", "dbo" or name of the database. If you are working with non-relational source, you can provide a dummy value here. server-name: name of the server created above to import metadata from. repository-name: Custom/extended "named" repositories from which metadata can be imported. See MetadataRepository interface for more details. Teiid provides a built in type called "DDL-FILE" see example below. schema-name: The foreign schema name to import from - it’s meaning is up to the translator. import qualifications : using this you can limit your import of the Tables from foreign datasource specified to this list. options-clause : The "importer" properties that can be used to refine the import process behavior of the metadata. Each Translator defines a set of "importer" properties with their documentation or through extension properties.

The below example shows importing metadata from a PostgreSQL using server example above.

Example 6

```
-- import from native database
IMPORT FOREIGN SCHEMA public
    FROM SERVER pgsql
    INTO test

-- in archive based vdbs(.vdb) you can provide each schema in a separate file and
```

```
pull them in main vdb.ddl file as
IMPORT FOREIGN SCHEMA public
    FROM REPOSITORY DDL-FILE
    INTO test OPTIONS ("ddl-file" '/path/to/schema.ddl')
```

Tip	The example IMPORT SCHEMA can be used with any custom Metadata Repository, in the REPOSITORY {DDL-FILE}, DDL-FILE represents a particular type of repository.

The above command imports public.customers, public.orders tables using pgsql's connection into a VDB schema test.

Importing another Virtual Database (VDB Reuse)

If you like to import another VDB that is created into the current VDB, the following command cn be used to import all the metadata

```
IMPORT DATABASE {vdb-name} VERSION {version} [WITH ACCESS CONTROL]
```

Specifying the WITH ACCESS CONTROL also imports any Data Roles defined in the other database.

Create Schema Objects

Most DDL statements that affect schema objects need the schema to be explicitly set. To be able to establish the schema context you are working with use following command:

Example: Set Schema

```
SET SCHEMA {schema-name};
```

then you will be create/drop/alter schema objects for that schema.

Example: Schema Object Creation

```
SET SCHEMA test;
CREATE VIEW my_view AS SELECT 'HELLO WORLD';
```

Data Roles

Data roles, also called entitlements, are sets of permissions defined per VDB that dictate data access (create, read, update, delete). Data roles use a fine-grained permission system that Teiid will enforce at runtime and provide audit log entries for access violations. To read more about Data Roles and Permissions see Data Roles and Permissions

Here we will show DDL support to create these Data Roles and corresponding permissions.

BNF for Create Data Role

```
CREATE ROLE {data-role}
    [WITH JAAS ROLE {enterprise-role}(,{enterprise-role})*]
    [WITH ANY AUTHENTICATED]
```

data-role: Data role referenced in the VDB enterprise-role: Enterprise role(s) that this data-role represents WITH ANY AUTHENTICATED: When present, this data-role is given to any user who is valid authenticated user.

Example: Create Data Role

```
CREATE ROLE readWrite WITH JASS ROLE developer,analyst;

CREATE ROLE readOnly WITH ANY AUTHENTICATED;
```

Note	Roles must be defined as a structural component of the VDB. GRANT/REVOKE may then appear after all of the database structure has been defined.

See Permissions for more details on the permission system.

BNF for GRANT/REVOKE command

```
GRANT [<permission-types> (,<permission-types>)* ]
    ON (<grant-resource>)
    TO {data-role}

GRANT (TEMPORARY TABLE | ALL PRIVILEGES)
    TO {data-role}

GRANT USAGE ON LANGUAGE {language-name}
    TO {data-role}

<permission-types> ::=
    SELECT | INSERT |  UPDATE | DELETE |
    EXECUTE | ALTER | DROP

<grant-resource> ::=
        TABLE {schema-name}.{table-name} [<condition>] |
        PROCEDURE {schema-name}.{procedure-name} [<condition>] |
        SCHEMA {schema-name} |
        COLUMN {schema-name}.{table-name}.{column-name} [MASK [ORDER n] {expression} ]

<condition> ::=
    CONDITION [CONSTRAINT] {boolean expression}

REVOKE [(<permission-types> (,<permission-types>)* )]
    ON (<revoke-resource>)
    FROM {data-role}

REVOKE
    (TEMPORARY TABLE | ALL PRIVILEGES)
    FROM {data-role}

REVOKE USAGE ON LANGUAGE {language-name}
    FROM {data-role}

<revoke-resource> ::=
        TABLE {schema-name}.{table-name} [CONDITION] |
        PROCEDURE {schema-name}.{procedure-name} [CONDITION] |
        SCHEMA {schema-name} |
        COLUMN {schema-name}.{table-name}.{column-name} [MASK]
```

- permission-types: Types of permissions to be granted
- language-name: Name of the language
- grant-resource: This is Schema element in the VDB on which this grant applies to.
- revoke-resource: This is Schema element in the VDB on which this revoke applies to. Specifying the CONDITION or MASK keyword will attempt to move the specific CONDITION or MASK for that resource.
- schema-name: Name of the schema this resource belongs to
- table-name: Name of the Table/View
- procedure-name: Procedure Name

- column-name: Name of the column
- condition: When present, the {expression} is appended to the WHERE clause of the query
- expression: any valid sql expression, this can include columns from referenced resource
- CONSTRAINT: When this is supplied along with CONDITION, the {boolean expression} is also applied during the INSERT/UPDATE queries. By default CONDITION **only** applies SELECT queries. Also CONSTRAINT does **NOT** apply to VIEWs only FOREIGN TABLES.

Warning	GRANT/REVOKE mostly function as direct replacements for the XML permission declarations. A grant/revoke has no effect on any other grant/revoke unless it represents the same resource, in which case its effect is combined.

Example: Give Read, write, update permission on single table to user with enterprise role "role1"

```
CREATE ROLE RoleA WITH JAAS ROLE role1;
...
GRANT INSERT, READ, UPDATE ON TABLE test.Customer TO RoleA;
```

Example : Give all permissions to user with "admin" enterprise role

```
CREATE ROLE everything WITH JAAS ROLE admin;
...
GRANT ALL PRIVILEGES TO everything;
```

Example : Use of CONDITION, all users can see only Orders table contents amount < 1000

```
CREATE ROLE base-role WITH ANY AUTHENTICATED;
...
GRANT READ ON TABLE test.Orders CONDITION 'amount < 1000' TO base-role;
```

Example : Use of CONDITION, override previous example to more privileged user

```
GRANT READ ON TABLE test.Orders CONDITION 'amount < 1000 and amount >=1000' TO RoleA;
```

Example : Restricting rows, ROW BASED SECURITY

```
GRANT READ ON TABLE test.CustomerOrders CONDITION CONSTRAINT 'name = user()' TO RoleA;
```

In the above example, user() function returns the currently logged in user id, if that matches to the name column, only those rows will be returned. There are functions like hasRole('x') that can be used too.

Example : Column Masking, mask "amount for all users"

```
GRANT READ ON COLUMN test.Order.amount
    MASK 'xxxx'
    TO base-role;
```

Example : Column Masking, mask "amount for all users when amount > 1000"

```
GRANT READ ON COLUMN test.Order.amount
    MASK 'CASE WHEN amount > 1000 THEN 'xxxx' END'
    TO base-role;
```

Example : Column Masking, mask "amount for all users" except the calling user is equal to the user()

```
GRANT READ ON COLUMN test.Order.amount
    MASK 'xxxx'
    CONDITION 'customerid <> user()'
```

```
    TO base-role;
```

Differences with vdb.xml metadata

Using a .ddl file instead of a .xml file to define a vdb will result in differences in how metadata is loaded when using a full server deploymnet of Teiid.

Using a vdb.ddl file does not support: * metadata caching at the schema level - although this feature may be added later * metadata reload if a datasource is unavailable at deployment time * parallel loading of source metadata

All of same limitations affect all VDBs (regardless of .xml or .ddl) when using Teiid Embedded.

XML VDB

XML based metadata may be deployed in a single xml file deployment or a zip file containing at least the xml file. The contents of the xml file will be similar either way. See Developing a Virtual Database for a discussion of the .vdb zip packaging. The XML may embedded or reference DDL.

XML File Deployment

You can simply create a **SOME-NAME-vdb.xml** file. The XML file captures information about the VDB, the sources it integrate, and preferences for importing metadata. The format of the XML file need to adhere to *vdb-deployer.xml* file, which is available in the schema folder under the docs with the Teiid distribution.

Important	The VDB name pattern must adhere to "-vdb.xml" for the Teiid VDB deployer to recognize this file when deployed in Teiid Server.

Tip	if you have existing VDB in combination of XML & DDL format, you can migrate to all DDL version using the "teiid-convert-vdb.bat" or "teiid-convert-vdb.sh" utility in the "bin" directory of the installation.

XML File Format

Example VDB XML Template

```
<vdb name="${name}" version="${version}">

    <!-- Optional description -->
    <description>...</description>

    <!-- Optional connection-type -->
    <connection-type>...</connection-type>

    <!-- VDB properties -->
    <property name="${property-name}" value="${property-value}" />

    <!-- UDF defined in an AS module,  see Developers Guide -->
    <property name ="lib" value ="{module-name}"></property>

    <import-vdb name="..." version="..." import-data-policies="true|false"/>

    <!-- define a model fragment for each data source -->
    <model visible="true" name="${model-name}" type="${model-type}" >

        <property name="..." value="..." />

        <source name="${source-name}" translator-name="${translator-name}"
            connection-jndi-name="${deployed-jndi-name}">

        <metadata type="${repository-type}">raw text</metadata>

        <!-- additional metadata
        <metadata type="${repository-type}">raw text</metadata>
        -->
    </model>

   <!-- define a model with multiple sources - see Multi-Source Models -->
   <model name="${model-name}" path="/Test/Customers.xmi">
        <property name="multisource" value="true"/>
        . . .
        <source name="${source-name}"
```

```
            translator-name="${translator-name}" connection-jndi-name="${deployed-jndi-name}"/>
        <source . . . />
        <source . . . />
    </model>

    <!-- see Reference Guide - Data Roles -->
    <data-role name="${role-name}">
        <description>${role-description}</description>
        ….
    </data-role>

    <!-- create translator instances that override default properties -->
    <translator name="${translator-name}" type="${translator-type}" />
        <property name="..." value="..." />
    </translator>
</vdb>
```

Note	**Property Substitution** - If a -vdb.xml file has defined property values like ${my.property.name.value}, these can be replaced by actual values that are defined through JAVA system properties. To define system properties on a WildFly server, please consult WildFly documentation.

Warning	You may choose to locally name vdb artifacts as you wish, but the runtime names of deployed VDB artifacts must either be *.vdb for a zip file or *-vdb.xml for an xml file. Failure to name the deployment properly will result in a deployment failure as the Teiid subsystem will not know how to properly handle the artifact.

VDB Element

Attributes

- *name*

The name of the VDB. The VDB name referenced through the driver or datasource during the connection time.

- *version*

The version of the VDB. Provides an explicit versioning mechanism to the VDB name - see VDB Versioning.

Description Element

Optional text element to describe the VDB.

Connection Type Element

Determines how clients can connect to the VDB. Can be one of BY_VERSION, ANY, or NONE. Defaults to BY_VERSION. See VDB Versioning.

Properties Element

see VDB Properties for properties that can be set at VDB level.

import-vdb Element

VDBs may reuse other VDBs deployed in the same server instance by using an "import-vdb" declaration in the vdb.xml file. An imported VDB can have it’s tables and procedures referenced by views and procedures in the importing VDB as if they are part of the VDB. Imported VDBs are required to exist before an importing VDB may start. If an imported VDB is undeployed, then any importing VDB will be stopped.+

An imported VDB includes all of its models and may not conflict with any model, data policy, or source already defined in the importing VDB. Once a VDB is imported it is mostly operationally independent from the base VDB. Only cost related metadata may be updated for an object from an imported VDB in the scope of the importing VDB. All other updates must be made through

the original VDB, but they will be visible in all imported VDBs. Even materialized views are separately maintained for an imported VDB in the scope of each importing VDB.

Example reuse VDB XML

```
<vdb name="reuse" version="1">
    <import-vdb name="common" version="1" import-data-policies="false"/>
    <model visible="true" type="VIRTUAL" name="new-model">
         <metadata type = "DDL"><![CDATA[
              CREATE VIEW x (
                y varchar
                ) AS
                  select * from old-model.tbl;
         ]]>
         </metadata>
    </model>
</vdb>
```

Attributes

- *name*

The name of the VDB to be imported.

- *version*

The version of the VDB to be imported (should be an positive integer).

- *import-data-policies*

Optional attribute to indicate whether the data policies should be imported as well. Defaults to "true".

Model Element

Attributes

- *name*

The name of the model is used as a top level schema name for all of the metadata imported from the connector. The name should be unique among all Models in the VDB and should not contain the '.' character.

- *visible*

By default this value is set to "true", when the value is set to "false", this model will not be visible to when JDBC metadata queries. Usually it is used to hide a model from client applications that should not directly issue queries against it. However, this does not prohibit either client application or other view models using this model, if they knew the schema for this model.

Property Elements

All properties are available as extension metadata on the corresponding `Schema` object that is accessible via the metadata API.

- *cache-metadata*

Can be "true" or "false". defaults to "false" for -vdb.xml deployments otherwise "true". If "false", Teiid will obtain metadata once for every launch of the vdb. "true" will save a file containing the metadata into the PROFILE/data/teiid directory Can be used to override the vdb level cache-metadata property.

- *teiid_rel:DETERMINISM*

Can be one of: DETERMINISM NONDETERMINISTIC COMMAND_DETERMINISTIC SESSION_DETERMINISTIC USER_DETERMINISTIC VDB_DETERMINISTIC DETERMINISTIC

Will influence the cache scope for result set cache entries formed from accessing this model. Alternatively the scope may be influenced through the Translator API or via table/procedure extension metadata.

Source Element

A source is a named binding of a translator and connection source to a model.

- *name*

The name of the source to use for this model. This can be any name you like, but will typically be the same as the model name. Having a name different than the model name is only useful in multi-source scenarios. In multi-source, the source names under a given model must be unique. If you have the same source bound to multiple models it may have the same name for each. An exception will be raised if the same source name is used for different sources.

- *translator-name*

The name or type of the Teiid Translator to use. Possible values include the built-in types (ws, file, ldap, oracle, sqlserver, db2, derby, etc.) and translators defined in the translators section.

- *connection-jndi-name*

The JNDI name of this source's connection factory. There should be a corresponding datasource that defines the connection factory in the JBoss AS. Check out the deploying VDB dependencies section for info. You also need to define these connection factories before you can deploy the VDB.

Property Elements

- *importer.<propertyname>*

Property to be used by the connector importer for the model for purposes importing metadata. See possible property name/values in the Translator specific section. Note that using these properties you can narrow or widen the data elements available for integration.

Metadata Element

The optional metadata element defines the metadata repository type and optional raw metadata to be consumed by the metadata repository.

- *type*

The metadata repository type. Defaults to NATIVE for source models. For all other deployments/models a value must be specified. Built-in types include DDL, NATIVE, and DDL-FILE. The usage of the raw text varies with the by type. NATIVE metadata repositories do not use the raw text. The raw text for DDL is expected to be be a series of DDL statements that define the schema. Note that, since <model> element means schema, you only use Schema Object DDL. The rest of the DDL statements can **NOT** be used in the artifact mode, as those constructs are defined by the XML file. Like <Model> element is similar to "CREATE SCHEMA …". Due to backwards compatibility Teiid supports both modes as both have their advantages.

DDL-FILE (used only with zip deployments) is similar to DDL, except that the raw text specifies an absolute path relative to the vdb root of the location of a file containing the DDL. See Metadata Repositories for more information and examples.

The INDEX type from Designer VDBs is deprecated.

Translator Element

Attributes

- *name*

The name of the the Translator. Referenced by the source element.

- *type*

The base type of the Translator. Can be one of the built-in types (ws, file, ldap, oracle, sqlserver, db2, derby, etc.).

Property Elements

- Set a value that overrides a translator default property. See possible property name/values in the Translator specific section.

VDB Reuse

VDBs may reuse other VDBs deployed in the same server instance by using an "import-vdb" declaration. An imported VDB can have it's tables and procedures referenced by views and procedures in the importing VDB as if they are part of the VDB. Imported VDBs are required to exist before an importing VDB may start. If an imported VDB is undeployed, then any importing VDB will be stopped.

An imported VDB includes all of its models and may not conflict with any model, data policy, or source already defined in the importing VDB. Once a VDB is imported it is mostly operationally independent from the base VDB. Only cost related metadata may be updated for an object from an imported VDB in the scope of the importing VDB. All other updates must be made through the original VDB, but they will be visible in all imported VDBs. Even materialized views are separately maintained for an imported VDB in the scope of each importing VDB.

Example reuse VDB XML

```
<vdb name="reuse" version="1">

    <property name="imported-model.visible" value="false"/>

    <import-vdb name="common" version="1" import-data-policies="false"/>

    <model visible="true" type="VIRTUAL" name="new-model">
        <metadata type = "DDL"><![CDATA[
             CREATE VIEW x (
               y varchar
               ) AS
                 select * from imported-model.tbl;
        ]]>
        </metadata>
    </model>
</vdb>
```

In the above example the reuse VDB will have access to all of the models defined in the common VDB and adds in the "new-model". The visibility of imported models may be overridden via boolean vdb properties using the key model.visible - shown above as imported-model.visible with a value of false.

Virtual Database Related Properties

Properties DATABASE Level

- *domain-ddl*
- *schema-ddl*
- *cache-metadata*

Can be "true" or "false". defaults to "false" for -vdb.xml deployments otherwise "true". If "false", Teiid will obtain metadata once for every launch of the vdb. "true" will save a file containing the metadata into the PROFILE/data/teiid directory

- *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.

- *lib*

Set to a list of modules for the vdb classpath for user defined function loading. See also Support for User-Defined Functions (Non-Pushdown).

- *security-domain*

Set to the security domain to use if a specific security domain is applicable to the VDB. Otherwise the security domain list from the transport will be used.

```
    <property name="security-domain" value="custom-security" />
```

Note	An admin needs to configure a matching "custom-security" login module in standalone-teiid.xml configuration file before the VDB is deployed.

- *connection.XXX*

For use by the ODBC transport and OData to set default connection/execution properties. See Driver Connection for all properties. Note these are set on the connection after it has been established.

```
    <property name="connection.partialResultsMode" value="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 of 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 of 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 Teiid server member in a cluster, and not cluster wide. Derived sessions, 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, PG metadata is always added to VDB unless System Properties set property *org.teiid.addPGMetadata* to false. This property enables adding PG metadata per VDB. 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 - see Internal Materialization. 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.

Properties Schema/Model Level

- *visible*

Marks the Schema is visible when value is *true*. visible falg is set to *false*, the Schema's metadata is hidden from any metadata requests. However note that this does not prohibit the user from issuing the queries against this Schema, in order to control the queries look into Data Roles.

- *multisource*

Marks the Schema as multi-source mode, where the data exists in partitions in multiple different sources. It is assumed that metadata of the Schema across all the data sources is exactly same.

- *multisource.columnName*

In a multi-source schema all the tables will be implicitly added with additional column to designate the partition column about identity of that source. This property defines the name of that column, the type will be always 'String'.

- *multisource.addColumn*

This flag to indicate, to add the implicit partition column to all the tables in this Schema. *true* value adds the column. Default is *false*.

- *allowed-languages*

The allowed-languages property enables the languages use for any purpose in the vdb, while the allow-language permission allows the language to be used by users with RoleA.

DDL Metadata for Schema Objects

The DDL for schema objects is common to both XML and DDL VDBs.

Table of Contents

- Data Types
- Creating a Foreign Table
  - Defining Table CONSTRAINTS
  - ALTER TABLE
- Create View
  - ALTER TABLE
  - INSTEAD OF TRIGGERS On VIEW (Update VIEW)
  - AFTER TRIGGERS On Source Tables
- Create Procedure/Function
- Extension Metadata
  - Built-in Namespace Prefixes

Data Types

The BNF for Data Types refer to Data Types

Creating a Foreign Table

A *FOREIGN* table is table that is defined on source schema that represents a real relational table in source databases like Oracle, SQLServer etc. For relational databases, Teiid has capability to automatically retrieve the database schema information upon the deployment of the VDB, if one like to auto import the existing schema. However, user can use below FOREIGN table semantics, when they would like to explicitly define tables on PHYSICAL schema or represent non-relational data as relational in custom translators.

BNF for Create Table

```
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>]*
```

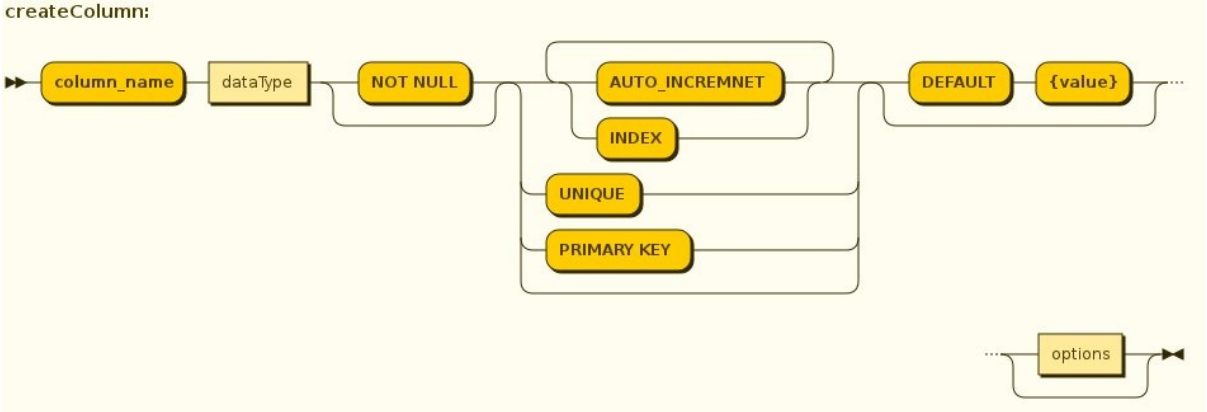
For validating BNF for create table refer to CREATE TABLE

Example 7: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 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 for View
CARDINALITY	int	Costing information. Number of rows in the table. Used for planning purposes
UPDATABLE	'TRUE'|'FALSE'	Defines if the view is allowed to update or not
ANNOTATION	string	Description of the view
DETERMINISM	NONDETERMINISTIC, COMMAND_DETERMINISTIC, SESSION_DETERMINISTIC, USER_DETERMINISTIC, VDB_DETERMINISTIC, DETERMINISTIC	Only checked on source tables



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

Property	Data Type or Allowed Values	Descriptio
UUID	string	A unique identifier fo the column
NAMEINSOURCE	string	If this is a column name on th FOREIGN table, this value represents name of the column in source database, if omitted the column name is use when querying fo data agains the source
CASE_SENSITIVE	'TRUE'|'FALSE'	
SELECTABLE	'TRUE'|'FALSE'	TRUE whe this columr is available for selectio from the us query
UPDATABLE	'TRUE'|'FALSE'	Defines if the column is updatabl Defaults to true if the view/table updatable.
SIGNED	'TRUE'|'FALSE'	
CURRENCY	'TRUE'|'FALSE'	
FIXED_LENGTH	'TRUE'|'FALSE'	
SEARCHABLE	'SEARCHABLE'|'UNSEARCHABLE'|'LIKE_ONLY'|'ALL_EXCEPT_LIKE'	column searchabilit usually dictated by the data typ
MIN_VALUE		
MAX_VALUE		

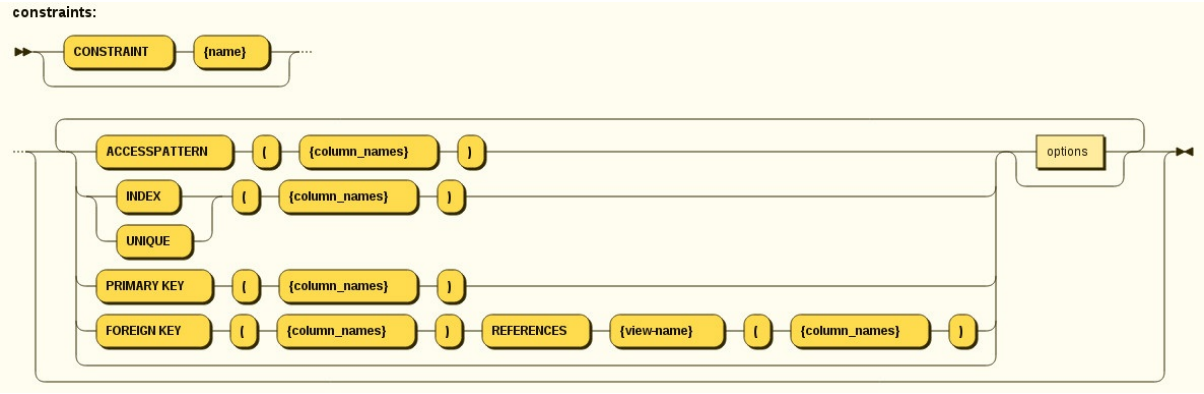
CHAR_OCTET_LENGTH	integer	
ANNOTATION	string	
NATIVE_TYPE	string	
RADIX	integer	
NULL_VALUE_COUNT	long	costing information Number of NULLS in this column
DISTINCT_VALUES	long	costing information Number of distinct values in the column

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

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

Defining Table CONSTRAINTS

Constraints can be defined on table/view to define indexes and relationships to other tables/views. This information is used by the Teiid 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

The BNF for ALTER table, refer to ALTER TABLE

Using the ALTER COMMAND, one can Add, Change, Delete columns, modify any OPTIONS values, and add constraints. Some examples below.

```
-- 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);
```

Create View

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 will be derived from the projected columns of the view's select transformation that is defined after the *AS* keyword.

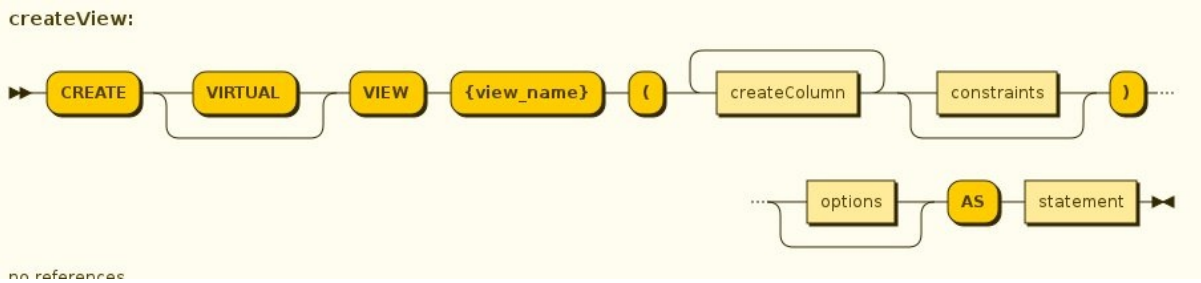
You can add functions, JOIN statements and WHERE clauses to a view data as if the data were coming from one single table.

Access patterns are not currently meaningful to views, but are still allowed by the grammar. Other constraints on views are also not enforced - unless 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>]*
```



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), they can also defined inline but if they are defined they can be only altered to modify their properties, you can not ADD or DROP new columns.

ALTER TABLE

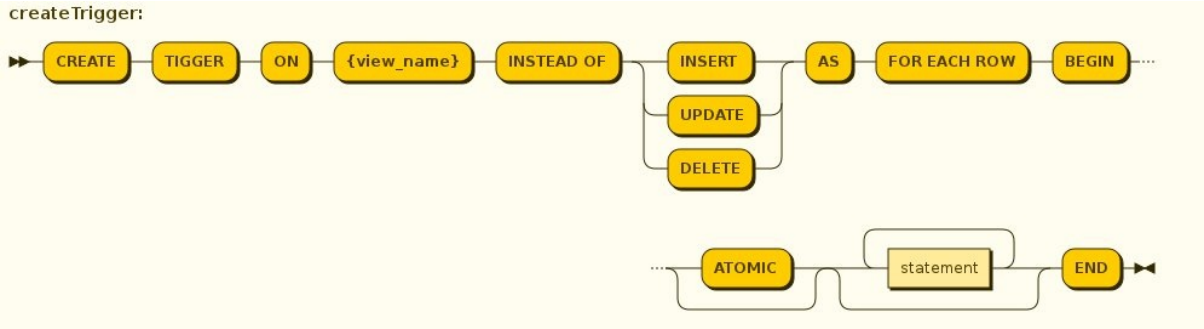
The BNF for ALTER VIEW, refer to ALTER TABLE

Using the ALTER COMMAND you can change the transformation query of the VIEW. You are **NOT** allowed to Alter the column information. Also the transformation query 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 support inserts, updates and 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 detailed explanation of these update procedures please refer to 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
```

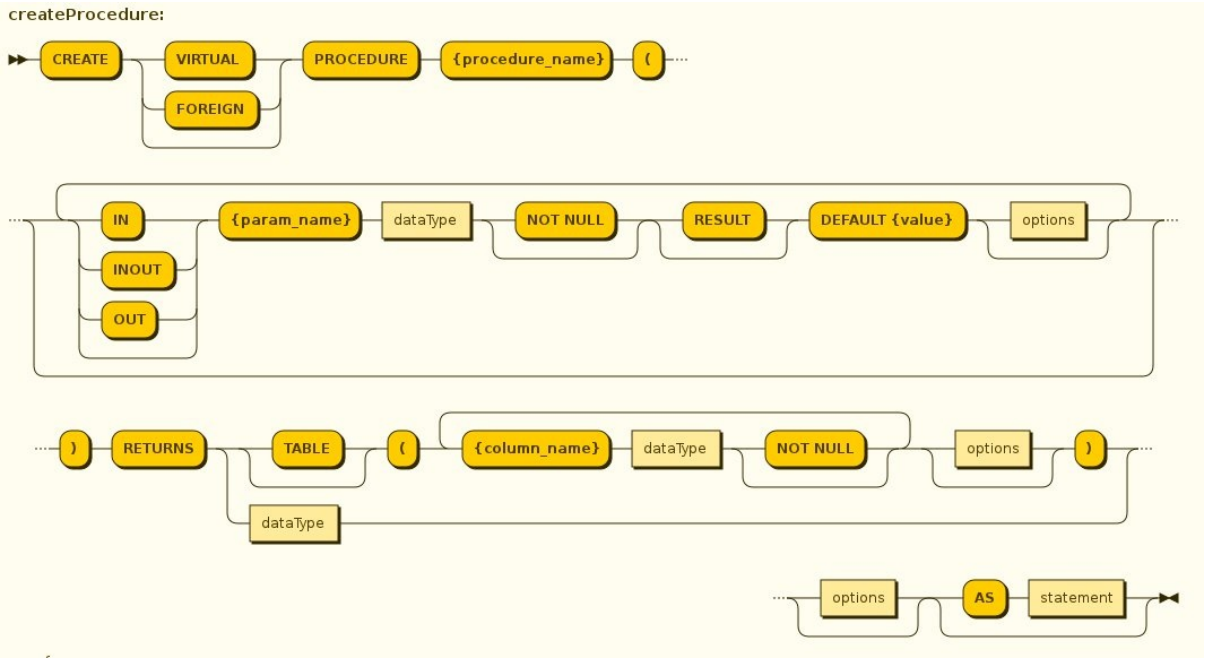
You will typically define a handler for each operation - INSERT/UPDATE/DELTE.

For more detailed explanation of these update procedures please refer to Update Procedures

Create Procedure/Function

Using the below syntax, user can define a

- Source Procedure ("CREATE FOREIGN PROCEDURE") - a stored procedure in source
- Source Function ("CREATE FOREIGN FUNCTION") - A function that is supported by the source, where Teiid will pushdown to source instead of evaluating in Teiid engine
- Virtual Procedure ("CREATE VIRTUAL PROCEDURE") - Similar to stored procedure, however this is defined using the Teiid's Procedure language and evaluated in the Teiid's engine.
- Function/UDF ("CREATE VIRTUAL FUNCTION") - A user defined function, that can be defined using the Teiid procedure language or can have the implementation defined using a JAVA Class.



See the full grammar for create function/procedure in the BNF for SQL Grammar.

Variable Argument Support

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
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. See also Source Supported Functions for creating FOREIGN functions that are supported by a source.

AGGREGATE FUNCTION OPTIONS:

Property	Data Type or Allowed Values	Description
ANALYTIC	'TRUE'|'FALSE'	indicates the aggregate function must be windowed. default false.
ALLOWS-ORDERBY	'TRUE'|'FALSE'	indicates the aggregate function supports an ORDER BY clause. default false
ALLOWS-DISTINCT	'TRUE'|'FALSE'	indicates the aggregate function supports the DISTINCT keyword. default false
DECOMPOSABLE	'TRUE'|'FALSE'	indicates the single argument aggregate function can be decomposed as agg(agg(x)) over subsets of data. default false

USES-DISTINCT-ROWS	'TRUE'|'FALSE'	indicates the aggregate function effectively uses distinct rows rather than all rows. default 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. To configure the Java library as dependency to the VDB, see Support for User-Defined Functions

PROCEDURE 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	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
```

Read more information about virtual procedures at Virtual Procedures, and these procedures are written using Procedure Language

Example:Define Virtual Function

```
CREATE VIRTUAL FUNCTION CustomerRank(customerid integer)
   RETURNS integer AS
   BEGIN
   ...
   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 Teiid, 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 Teiid 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 parens. See also Translators#native
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');
```

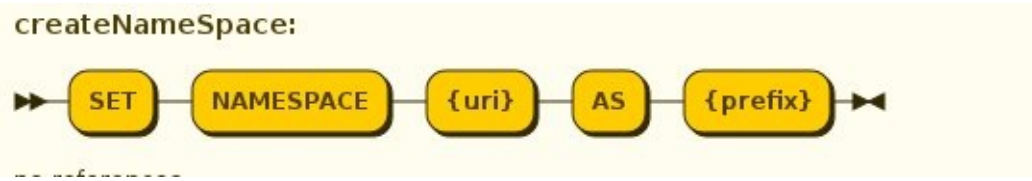
Example:Sequence Native Query

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

Tip	Until Teiid provides higher-level metadata support for sequences, a source function representation is the best fit 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 define namespace for the properties such that they will not collide with the Teiid specific properties. The property should be prefixed with alias of the Namespace. Prefixes starting with teiid_ are reserved for use by Teiid.



Example of Namespace

```
SET NAMESPACE 'http://custom.uri' AS foo

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

Built-in Namespace Prefixes

Prefix	URI	Description

teiid_rel	http://www.teiid.org/ext/relational/2012	Relational extensions. Uses include function and native query metadata
teiid_sf	http://www.teiid.org/translator/salesforce/2012	Salesforce extensions.
teiid_mongo	http://www.teiid.org/translator/mongodb/2013	MongoDB Extensions
teiid_odata	http://www.jboss.org/teiiddesigner/ext/odata/2012	OData Extensions
teiid_accumulo	http://www.teiid.org/translator/accumulo/2013	Accumulo extensions
teiid_excel	http://www.teiid.org/translator/excel/2014	Excel Extensions
teiid_ldap	http://www.teiid.org/translator/ldap/2015	LDAP Extensions
teiid_rest	http://teiid.org/rest	REST Extensions
teiid_pi	http://www.teiid.org/translator/pi/2016	PI Database Extensions

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.

The DDL for domains is common to both XML and DDL VDBs. However in an XML vdb domains must be defined in a VDB property "domain-ddl".

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.

DDL VDB Example

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

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

XML VDB Example

```
<vdb name="Portfolio" version="1">

    <property name="domain-ddl" value="CREATE DOMAIN ssn AS VARCHAR(9); CREATE DOMAIN myint AS integer not null
;" />
    ...
```

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

Limitations

Domain names are not yet recognized in every place that a data type is expected, such as in:

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

The ODBC/pg metadata will show the base type name, rather than the domain name when querying pg_attribute.

Multisource Models

Multisource models can be used to quickly access data in multiple sources with homogeneous metadata. When you have multiple instances using identical schema (horizontal sharding), Teiid can help you gather data across all the instances, using "multisource" models. In this scenario, instead of creating/importing a model for every data source, one source model is defined to represents the schema and is configured with multiple data "sources" underneath it. During runtime when a query issued against this model, the query engine analyzes the information and gathers the required data from all sources configured and gathers the results and provides in a single result. Since all sources utilize the same physical metadata, this feature is most appropriate for accessing the same source type with multiple instances.

Configuration

To mark a model as multisource, the model property *multisource* can be set to true or more than one source can be listed for the model in the "vdb.xml" file. Here is a code example showing a vdb with single model with multiple sources defined.

```
<vdb name="vdbname" version="1">
    <model visible="true" type="PHYSICAL" name="Customers" path="/Test/Customers.xmi">
        <property name="multisource" value="true"/>
        <!-- optional properties
        <property name="multisource.columnName" value="somename"/>
        <property name="multisource.addColumn" value="true"/>
        -->
        <source name="chicago"
            translator-name="oracle" connection-jndi-name="chicago-customers"/>
        <source name="newyork"
            translator-name="oracle" connection-jndi-name="newyork-customers"/>
        <source name="la"
            translator-name="oracle" connection-jndi-name="la-customers"/>
    </model>
</vdb>
```

NOTE Tooling support for managing the multisource feature is limited. You must deploy a separate data source for each source defined in the xml file.

In the above example, the VDB has a single model called `Customers`, that has multiple sources (`chicago`, `newyork`, and `la`) that define different instances of data.

The Multisource Column

When a model is marked as multisource, the engine will add or use an existing column on each table to represent the source name values. In the above vdb.xml the column would return `chicago`, `la`, `newyork` for each of the respective sources. The name of the column defaults to SOURCE_NAME, but is configurable by setting the model property **multisource.columnName**. If a column already exists on the table (or an IN procedure parameter) with the same name, the engine will assume that it should represent the multisource column and it will not be used to retrieve physical data. If the multisource column is not present, the generated column will be treated as a pseudo column which is not selectable via wildcards (* nor tbl.*).

This allows queries like the following:

```
select * from table where SOURCE_NAME = 'newyork'
update table column=value where SOURCE_NAME='chicago'
delete from table where column = x and SOURCE_NAME='la'
insert into table (column, SOURCE_NAME) VALUES ('value', 'newyork')
```

The Multi-Source Column in System Metadata

The pseudo column is by default not present in your actual metadata; it is not added on source tables/procedures when you import the metadata. If you would like to use the multisource column in your transformations to control which sources are accessed or updated and/or want the column reported via metadata facilities, there are several options:

- If directly using DDL, the pseduo-column will already be available to transformations, but will not be present in your System metadata by default. If using DDL and want to be selective (rather than using the **multisource.addColumn** property), you can manually add the column via DDL.
- With either VDB type to make the multisource column present in the system metadata, you may set the model property **multisource.addColumn** to true on a multisource model. If the table has a column or the procedure has a parameter already with a matching name, then an additional column will not be added. A variadic procedure can still have a source parameter added, but it can only be specified when using named parameters. Care should be taken though when using this property as any transformation logic (views/procedures) that you have defined will not have been aware of the multisource column and may fail validation upon server deployment.
- You can manually add the multisource column.

Other Partitioning Columns

If other columns on a multisource table are partitioned across the sources, the optimizer can be made aware via an extension property. Operations over that column, such as group by or distinct, can then be pushed separately to each source without post-processing in the engine. If you need to enable this, add the extension metadata property teiid_rel:multisource.partitioned=true to the column.

Example DDL

```
CREATE FOREIGN TABLE TBL (my_col integer options ("teiid_rel:multisource.partitoned" true) ...);
```

Planning and Execution

The planner logically treats a multisource table as if it were a view containing the union all of the respective source tables. More complex partitioning scenarios, such as heterogeneous sources or list partitioning will require the use of a Federated Optimizations#Partitioned Union.

Most of the federated optimizations available over unions are still applicable in multisource mode. This includes aggregation pushdown/decomposition, limit pushdown, join partitioning, etc.

You can add/remove sources from multisource models at runtime with the admin addSource and removeSource options. The processing of a multisource plan will determine the set of multisource targets when the access node is opened. If the plan is reused and the sources change since the last execution, the multisource access will be regenerated. If a source is added after a relevant multisource query starts, it will not be in the results. If a source is removed after a relevant multisource query starts, it will be treated as a null source which should in most situations allow the query to complete normally.

That the SHOW PLAN output will vary upon when it is obtained. If you get the SHOW PLAN output prior to execution, the multisource access will appear as a single access node. After execution the SHOW PLAN output will show the set of sources accessed as individual nodes.

SELECTs, UPDATEs, DELETEs

- A multisource query against a SELECT/UPDATE/DELETE may affect any subset of the sources based upon the evaluation of the WHERE clause.
- The multisource column may not be targeted in an update change set.

- The sum of the update counts for UPDATEs/DELETEs will be returned as the resultant update count.
- When running under a transaction in a mode that detects the need for a transaction and multiple updates may performed or a transactional read is required and multiple sources may be read from, a transaction will be started to enlist each source.

INSERTs

- A multisource INSERT must use the source_name column as an insert column to specify which source should be targeted by the INSERT. Only an INSERT using the VALUES clause is supported.

Stored Procedures

A physical stored procedures requires the addition of a string in parameter matching the multisource column name to specify which source the procedure is executed on. If the parameter is not present and defaults to a null value, then the procedure will be executed on each source. It is not possible to execute procedures that are required to return IN/OUT, OUT, or RETURN parameters values on more than 1 source.

Example DDL

```
CREATE FOREIGN PROCEDURE PROC (arg1 IN STRING NOT NULL, arg2 IN STRING, SOURCE_NAME IN STRING)
```

Example Calls Against A Single Source

```
CALL PROC(arg1=>'x', SOURCE_NAME=>'sourceA')
EXEC PROC('x', 'y', 'sourceB')
```

Example Calls Against All Sources

```
CALL PROC(arg1=>'x')
EXEC PROC('x', 'y')
```

Metadata Repositories

Traditionally the metadata for a Virtual Database is supplied to Teiid engine through a VDB archive file. A number of *MetadataRepository* instances contribute to the loading of the metadata. Built-in metadata repositories include the following:

NATIVE

This is only applicable on source models (and is also the default), when used the metadata for the model is retrieved from the source database itself.

Sample vdb.xml file

```
<vdb name="{vdb-name}" version="1">
    <model name="{model-name}" type="PHYSICAL">
        <source name="AccountsDB" translator-name="oracle" connection-jndi-name="java:/oracleDS"/>
        <metadata type="NATIVE"></metadata>
    </model>
</vdb>
```

DDL

Sample vdb.xml file

```
<vdb name="{vdb-name}" version="1">
    <model name="{model-name}" type="PHYSICAL">
        <source name="AccountsDB" translator-name="oracle" connection-jndi-name="java:/oracleDS"/>
        <metadata type="DDL">
          **DDL Here**
        </metadata>
    </model>
</vdb>
```

This is applicable to both source and view models. See DDL Metadata for more information on how to use this feature.

DDL-FILE

Sample vdb.xml file

```
<vdb name="{vdb-name}" version="1">
    <model name="{model-name}" type="PHYSICAL">
        <source name="AccountsDB" translator-name="oracle" connection-jndi-name="java:/oracleDS"/>
        <metadata type="DDL-FILE">/accounts.ddl</metadata>
    </model>
</vdb>
```

DDL is applicable to both source and view models in zip VDB deployments. See DDL Metadata for more information on how to use this feature.

UDF (11.2+)

Sample vdb.xml file

```
<vdb name="{vdb-name}" version="1">
```

```
    <model name="{model-name}" type="VIRTUAL">
        <property name="importer.schemaName" value="org.foo.Class"/>
        <metadata type="UDF"></metadata>
    </model>
</vdb>
```

Sample ddl file

```
CREATE DATABASE {vdb-name} VERSION '1';
USE DATABASE {vdb-name} VERSION '1';
CREATE VIRTUAL SCHEMA {model-name};
IMPORT FOREIGN SCHEMA "org.foo.Class" FROM REPOSITORY UDF INTO {model-name};
```

The logic will import all static functions that return non-void results, or import the user defined aggregate function if the class implements the UserDefinedAggregate interface.

Chaining Repositories

When defining the metadata type for a model, multiple metadata elements can be used. All the repository instances defined are consulted in the order configured to gather the metadata for the given model. For example:

Sample vdb.xml file

```
<vdb name="{vdb-name}" version="1">
    <model name="{model-name}" type="PHYSICAL">
        <source name="AccountsDB" translator-name="oracle" connection-jndi-name="java:/oracleDS"/>
        <metadata type="NATIVE"/>
        <metadata type="DDL">
          **DDL Here**
        </metadata>
    </model>
</vdb>
```

Note	For the above model, *NATIVE* importer is first used, then DDL importer used to add additional metadata to *NATIVE* imported metadata.

Custom

See Custom Metadata Repository

REST Service Through VDB

With help of DDL Metadata variety of metadata can be defined on VDB schema models. This metadata is not limited to just defining the tables, procedures and functions. The capabilities of source systems or any extensions to metadata can also be defined on the schema objects using the OPTIONS clause. One such extension properties that Teiid defines is to expose Teiid procedures as REST based services.

Expose Teiid Procedure as Rest Service

One can define below REST based properties on a Teiid virtual procedure, and when the VDB is deployed the Teiid VDB deployer will analyze the metadata and deploy a REST service automatically. When the VDB un-deployed the REST service also deployed.

Property Name	Description	Is Required	Allowed Values
METHOD	HTTP Method to use	Yes	Method names including GET | POST| PUT | DELETE
URI	URI of procedure	Yes	A relative path, which can include parameters as {param name}. For example: /procedure/{param1}
PRODUCES	Type of content produced by the service	No. If not specified will be inferred from the procedure return value.	A comma separated list of the full MIME type(s), or one of xml, json, or plain
CHARSET	When string/xml data is returned, this will be the encoding	No. If not specified will default to the system default charset.	A valid Java charset name, such as UTF-8, US-ASCII, etc.

The above properties must be defined with NAMESPACE `http://teiid.org/rest' on the metadata. Here is an example VDB that defines the REST based service.

Example VDB with REST based metadata properties

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<vdb name="sample" version="1">
    <property name="{http://teiid.org/rest}auto-generate" value="true"/>

    <model name="PM1">
        <source name="text-connector" translator-name="loopback" />
         <metadata type="DDL"><![CDATA[
                CREATE FOREIGN TABLE G1 (e1 string, e2 integer);
                CREATE FOREIGN TABLE G2 (e1 string, e2 integer);
        ]]> </metadata>
    </model>
    <model name="View" type ="VIRTUAL">
         <metadata type="DDL"><![CDATA[
            SET NAMESPACE 'http://teiid.org/rest' AS REST;
            -- This procedure produces XML payload
            CREATE VIRTUAL PROCEDURE g1Table(IN p1 integer) RETURNS TABLE (xml_out xml) OPTIONS (UPDATECOUNT 0,
 "REST:METHOD" 'GET', "REST:URI" 'g1/{p1}')
            AS
            BEGIN
                SELECT XMLELEMENT(NAME "rows", XMLATTRIBUTES (g1Table.p1 as p1), XMLAGG(XMLELEMENT(NAME "row",
```

```
XMLFOREST(e1, e2)))) AS xml_out FROM PM1.G1;
            END

            -- This procedure produces JSON payload
            CREATE VIRTUAL PROCEDURE g2Table(IN p1 integer) RETURNS TABLE (json_out clob) OPTIONS (UPDATECOUNT
0, "REST:METHOD" 'GET', "REST:URI" 'g2/{p1}')
            AS
            BEGIN
                SELECT JSONOBJECT(JSONARRAY_AGG(JSONOBJECT(e1, e2)) as g2) AS json_out FROM PM1.G2;
            END
            ]]> </metadata>
    </model>

</vdb>
```

Note	<property name="{http://teiid.org/rest}auto-generate" value="true"/>, can be used to control the generation of the REST based WAR based on the VDB. This property along with at least one procedure with REST based extension metadata is required to generate a REST WAR file. Also, the procedure needs to return result set with single column of either XML, Clob, Blob or String. When PRODUCES property is not defined, this property is derived from the result column that is projected out. The PRODUCES values xml, json, and plain are actually converted to the MIME types application/xml, application/json, and text/plain respectively. You may enter the full MIME type if you need, such as text/html.

When the above VDB is deployed in the WildFly + Teiid server, and if the VDB is valid and after the metadata is loaded then a REST war generated automatically and deployed into the local WildFly server. The REST VDB is deployed with "{vdb-name}_{vdb-version}" context. The model name is prepended to uri of the service call. For example the procedure in above example can be accessed as

```
http://{host}:8080/sample_1/view/g1/123
```

where "sample_1" is context, "view" is model name, "g1" is URI, and 123 is parameter {p1} from URI. If you defined a procedure that returns a XML content, then REST service call should be called with "accepts" HTTP header of "application/xml". Also, if you defined a procedure that returns a JSON content and PRODUCES property is defined "json" then HTTP client call should include the "accepts" header of "application/json". In the situations where "accepts" header is missing, and only one procedure is defined with unique path, that procedure will be invoked. If there are multiple procedures with same URI path, for example one generating XML and another generating JSON content then "accepts" header directs the REST engine as to which procedure should be invoked to get the results. A wrong "accepts" header will result in error.

"GET Methods"

When designing the procedures that will be invoked through GET based call, the input parameters for procedures can be defined in the PATH of the URI, as the {p1} example above, or they can also be defined as query parameter, or combination of both. For example

```
http://{host}:8080/sample_1/view/g1?p1=123
http://{host}:8080/sample_1/view/g1/123?p2=foo
```

Make sure that the number of parameters defined on the URI and query match to the parameters defined on procedure definition. If you defined a default value for a parameter on the procedure, and that parameter going to be passed in query parameter on URL then you have choice to omit that query parameter, if you defined as PATH you must supply a value for it.

"POST methods"

'POST' methods MUST not be defined with URI with PATHS for parameters as in GET operations, the procedure parameters are automatically added as @FormParam annotations on the generated procedure. A client invoking this service must use FORM to post the values for the parameters. The FORM field names MUST match the names of the procedure parameters names.

If any one of the procedure parameters are BLOB, CLOB or XML type, then POST operation can be only invoked using "multipart/form-data" RFC-2388 protocol. This allows user to upload large binary or XML files efficiently to Teiid using streaming".

"VARBINARY type"

If a parameter to the procedure is VARBINARY type then the value of the parameter must be properly BASE64 encoded, irrespective of the HTTP method used to execute the procedure. If this VARBINARY has large content, then consider using BLOB.

Security on Generated Services

By default all the generated Rest based services are secured using "HTTPBasic" with security domain "teiid-security" and with security role "rest". However, these properties can be customized by defining the then in vdb.xml file.

Example vdb.xml file security specification

```
<vdb name="sample" version="1">
    <property name="{http://teiid.org/rest}auto-generate" value="true"/>
    <property name="{http://teiid.org/rest}security-type" value="HttpBasic"/>
    <property name="{http://teiid.org/rest}security-domain" value="teiid-security"/>
    <property name="{http://teiid.org/rest}security-role" value="example-role"/>
    ...
</vdb>
```

- *security-type* - defines the security type. allowed values are "HttpBasic" or "none". If omitted will default to "HttpBasic"
- *security-domain* - defines JAAS security domain to be used with HttpBasic. If omitted will default to "teiid-security"
- *security-role* - security role that HttpBasic will use to authorize the users. If omitted the value will default to "rest"

Note	**rest-security** - it is our intention to provide other types of securities like Kerberos and OAuth2 in future releases.

Special Ad-Hoc Rest Services

Apart from the explicitly defined procedure based rest services, the generated jax-rs war file can also include a special rest based service under URI "/query" that can take any XML or JSON producing SQL as parameter and expose the results of that query as result of the service.

The model/schema must be have the {http://teiid.org/rest}security-role property set to true to expose the procedure.

This service is defined with "POST", accepting a Form Parameter named "sql". For example, after you deploy the VDB defined in above example, you can issue a HTTP POST call as

```
    http://localhost:8080/sample_1/view/query
    sql=SELECT XMLELEMENT(NAME "rows",XMLAGG(XMLELEMENT(NAME "row", XMLFOREST(e1, e2)))) AS xml_out FROM PM1.G1
```

A sample HTTP Request from Java can be made like below

```
   public static String httpCall(String url, String method, String params) throws Exception {
        StringBuffer buff = new StringBuffer();
        HttpURLConnection connection = (HttpURLConnection) new URL(url).openConnection();
        connection.setRequestMethod(method);
        connection.setDoOutput(true);

        if (method.equalsIgnoreCase("post")) {
            OutputStreamWriter wr = new OutputStreamWriter(connection.getOutputStream());
```

```
            wr.write(params);
            wr.flush();
        }

        BufferedReader serverResponse = new BufferedReader(new InputStreamReader(connection.getInputStream()));
        String line;
        while ((line = serverResponse.readLine()) != null) {
            buff.append(line);
        }
        return buff.toString();
    }

    public static void main(String[] args) throws Exception {
        String params = URLEncoder.encode("sql", "UTF-8") + "=" + URLEncoder.encode("SELECT XMLELEMENT(NAME "ro
ws",XMLAGG(XMLELEMENT(NAME "row", XMLFOREST(e1, e2)))) AS xml_out FROM PM1.G1", "UTF-8");
        httpCall("http://localhost:8080/sample_1/view/query", "POST", params);
    }
```

VDB Reuse

VDBs may reuse other VDBs deployed in the same server instance by using an "import-vdb" declaration in the vdb.xml file. An imported VDB can have it's tables and procedures referenced by views and procedures in the importing VDB as if they are part of the VDB. Imported VDBs are required to exist before an importing VDB may start. If an imported VDB is undeployed, then any importing VDB will be stopped.

Once a VDB is imported it is mostly operationally independent from the base VDB. Only cost related metadata may be updated for an object from an imported VDB in the scope of the importing VDB. All other updates must be made through the original VDB, but they will be visible in all imported VDBs. Even materialized views are separately maintained for an imported VDB in the scope of each importing VDB.

Example reuse VDB XML

```
<vdb name="reuse" version="1">

    <property name="imported-model.visible" value="false"/>

    <import-vdb name="common" version="1" import-data-policies="false"/>

    <model visible="true" type="VIRTUAL" name="new-model">
         <metadata type = "DDL"><![CDATA[
              CREATE VIEW x (
                y varchar
                ) AS
                  select * from imported-model.tbl;
         ]]>
         </metadata>
    </model>
</vdb>
```

In the above example the reuse VDB will have access to all of the models defined in the common VDB and adds in the "new-model". The visibility of imported models may be overridden via boolean vdb properties using the key model.visible - shown above as imported-model.visible with a value of false.

An imported VDB includes all of its models and may not conflict with any model, data policy, or source already defined in the importing VDB. The import logic though does recognize imported VDBs that perform nothing but imports and will instead import only distinct imports.

Common Example

```
<code>
<vdb name="OneVDB" version="1">
    <description>One VDB</description>

    <import-vdb name=CommonVDB" version="1"/>
    <import-vdb name="OtherVDB" version="1"/>

</code>

<code>
<vdb name="TwoVDB" version="1">
    <description>TwoVDB</description>

    <import-vdb name=CommonVDB" version="1"/>
    <import-vdb name="SomeOtherVDB" version="1"/>

</code>

<code>
<vdb name="ThirdVDB" version="1">
```

```
    <description>Third VDB</description>

    <import-vdb name=OneVDB" version="1"/>
    <import-vdb name="TwoVDB" version="1"/>

</code>
```

In the above example CommonVDB will only be imported a single time by ThirdVDB, since the import logic recognizes that the importing VDBs perform nothing but imports themselves.

SQL Support

Teiid 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 SQL's usage within Teiid. See the BNF for SQL Grammar for the exact form of SQL accepted by Teiid.

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 it’s be escaped with an additional double quote. e.g. "some", " id"
- Because different data sources organize tables in different ways, some prepending catalog or schema or user information, Teiid 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."sourcescema.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 Teiid.

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 '.' 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

Teiid’s reserved words include the standard SQL 2003 Foundation, SQL/MED, and SQL/XML reserved words, as well as Teiid specific words such as BIGINTEGER, BIGDECIMAL, or MAKEDEP. See the BNF for SQL Grammar Reserved Keywords and Reserved Keywords For Future Use sections for all reserved words.

Expressions

Identifiers, literals, and functions can be combined into expressions. Expressions can be used almost anywhere in a query – SELECT, FROM (if specifying join criteria), WHERE, GROUP BY, HAVING, or ORDER BY.

Teiid supports the following types of expressions:

- Column Identifiers
- Literals
- Scalar Functions
- Aggregate Functions
- Window Functions
- Case and Searched Case
- Scalar Subqueries
- Parameter References
- Criteria
- Arrays

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.

Literals

Literal values represent fixed values. These can any of the 'standard' 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'
```

escaped single tick

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

```
5
```

scientific notation

```
-37.75e01
```

exact numeric type BigDecimal

```
100.0
```

```
true
```

```
false
```

unicode character

```
'\u0027'
```

binary

```
X'0F0A'
```

Date/Time Literals can use either JDBC Escaped Literal Syntax:

Date Literal

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

Time Literal

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

Timestamp Literal

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

Or the ANSI keyword syntax:

Date Literal

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

Time Literal

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

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.

Teiid supports the following aggregate functions:

- 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. See also the JSONArray function.

integer value example

```
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.

string agg example

```
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 convertable to string, the delim if specified must be a literal, and the order by is required. This is just a parsing alias for an equivalent string_agg expression.

list agg example

```
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 matching 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.

Window Functions

Teiid supports 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
```

aggregate can be any Aggregate Functions. Keywords exist for the 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.

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. The is no guaranteed affect 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.

Teiid will process all window functions with the same window specification together. In general a full pass over the row values coming into the SELECT clause will be required for each unique window specification. For each window specification the values will be 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, has 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
John	60000	100000	3	3	3
Suzie	60000	150000	3	3	4
Suzie	150000	150000	1	1	1

Case and Searched Case

Teiid supports two forms of the CASE expression which allows conditional logic in a scalar expression.

Supported forms:

- 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
```

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 other types of subqueries, see the Subqueries section.

Parameter References

Parameters are specified using a '?' symbol. Parameters may only be used with PreparedStatement or CallableStatements in JDBC. Each parameter is linked to a value specified by 1-based index in the JDBC API.

Arrays

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

empty arrays

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

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 parenthesis, rather than a simple nested expression.

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.

Operator Precedence

Teiid parses and evaluates operators with higher precedence before those with lower precedence. Operator with equal precedence are left associative. Operator precedence listed from high to low:

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

Criteria

Criteria may be:

- Predicates that evaluate to true or false
- Logical criteria that combines criteria (AND, OR, NOT)
- A value expression with 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 equivalent and never produces an UNKNOWN value.

Note	Using IS DISTINCT FROM in a join predicate that is not pushed down will not yet produce as performant of a plan as a regular comparison as the optimizer is not tuned to handling IS DISTINCT FROM.

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

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

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

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	Teiid does not exhaustively validate SIMILAR TO pattern values. Rather the pattern is converted to an equivalent regular expression. Care should be taken not to rely on general regular expression features when using SIMILAR TO. If additional features are needed, then LIKE_REGEX should be used. Usage of a non-literal pattern is discouraged as pushdown support is limited.

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

LIKE_REGEX allows for standard regular expression syntax to be used 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 - see the java.util.regex.Pattern class for details.

Note	Teiid 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. Additional not all sources support the same regular expression flavor or extensions. Care should be taken in pushdown situations to ensure that the pattern used will have same meaning in Teiid and across all applicable sources.

```
EXISTS (subquery)
```

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

Teiid 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

Tip	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

Teiid parses and evaluates conditions with higher precedence before those with lower precedence. Conditions with equal precedence are left associative. Condition precedence listed 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 however that to prevent lookaheads the parser does not accept all possible criteria sequences. For example "a = b is null" is not accepted, since 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". See BNF for SQL Grammar for all parsing rules.

Scalar Functions

Teiid provides an extensive set of built-in scalar functions. See also SQL Support and Datatypes. In addition, Teiid provides the capability for user defined functions or UDFs. See the Developers Guide for adding UDFs. Once added UDFs may be called just like any other function.

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
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
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 Teiid 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

Teiid 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

Teiid 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'
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'

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 clob
ASCII(x)	Provide ASCII value of the left most character in x. The empty string will as input will return null.	return type is integer
CHR(x) CHAR(x)	Provide the character for ASCII value x [a]	x in {integer}
CONCAT(x, y)	Concatenates x and y with ANSI semantics. If x and/or y is null, returns null.	x, y in {string}
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
QUERYSTRING(path [, expr [AS name] …])	Returns a properly encoded query string appended to the given path. Null valued expressions are omitted, and a null path is treated as ''. Names are optional for column reference expressions.e.g. `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 specficed then the blank space ' ' is used.	x in {character}, y in {string}
UCASE(x)	Uppercase of x	x in {string}
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	x in {string}

	after an escape character, that character will appear in the output and the escape character will be ignored.	

[a] Non-ASCII range characters or integers used in these functions may produce different results or exceptions depending on where the function is evaluated (Teiid vs. source). Teiid'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 Teiid 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 do not support negative indexing. If any of these inconsistencies impact you, then please log an issue.

Encoding Functions

TO_CHARS

Return a clob from the 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] See the Charset docs for more on supported Charset names.

Replacement Functions

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	multiline	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')
```

Date_Time Functions

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

Parse and format Date/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 Teiid 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

EXTRACT(YEAR|MONTH|DAY |HOUR|MINUTE|SECOND FROM x)	Return the given field value from the date value x. Produces the same result as the associated YEAR, MONTH, DAYOFMONTH, HOUR, MINUTE, SECOND functions.The SQL specification also allows for TIMEZONE_HOUR and TIMEZONE_MINUTE as extraction targets. In Teiid all date values are in the timezone of the server.	x in {date, time, timestamp}, returns integer
FORMATDATE(x, y)	Format date x using format y	x is date, y is string, returns string
FORMATTIME(x, y)	Format time x using format y	x is time, y is string, returns string
FORMATTIMESTAMP(x, y)	Format timestamp x using format y	x is timestamp, y is string, returns string
FROM_MILLIS (millis)	Return the Timestamp value for the given milliseconds	long UTC timestamp in milliseconds
FROM_UNIXTIME (unix_timestamp)	Return the Unix timestamp as a String value with the default format of yyyy/mm/dd hh:mm:ss	long Unix timestamp (in seconds)
HOUR(x)	Return hour (in military 24-hour format)	x in {time, timestamp}, returns integer
MINUTE(x)	Return minute	x in {time, timestamp}, returns integer
MODIFYTIMEZONE (timestamp, startTimeZone, endTimeZone)	Returns a timestamp based upon the incoming timestamp adjusted for the differential between the start and end time zones. i.e. 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
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, see also System Properties for customization	x in {date, timestamp}, returns integer
YEAR(x)	Return four-digit year	x in {date, timestamp}, returns integer

Timestampadd/Timestampdiff

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

- Timestampdiff typically returns an integer, however Teiid's version returns a long. You may receive an exception if you expect a value out of the integer range from a pushed down timestampdiff.
- Teiid's implementation of timestamp diff in 8.2 and prior versions returned values based upon 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. See System Properties for backwards compatibility.

Parsing Date Datatypes from Strings

Teiid 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. You can learn more about how this class defines date and time string formats by visiting the Javadocs for SimpleDateFormat. Note that the format strings will be locale 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. Additionally, you can specify a custom time zone by GMT offset: GMT[+/-]HH:MM.

For example: GMT-05:00

Type Conversion Functions

Within your queries, you can convert between datatypes using the CONVERT or CAST keyword. See also Type Conversions

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

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.

Decode Functions

Decode functions allow you to have the Teiid 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 , is used. y has the formate SearchDelimResultDelimSearchDelimResult[DelimDefault] Returns Default if specified or x if there are no matches. **Deprecated. Use a CASE expression instead.**	all string
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 , is used. y has the formate 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 Teiid 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 Teiid 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 Teiid 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 Teiid Server found in that column.

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 Teiid System caches the result. The Teiid 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. See the Caching Guide for more on the caching aspects of the Lookup function.

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

System Functions

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

Table of Contents

- COMMANDPAYLOAD
- ENV
- ENV_VAR
- SYS_PROP
- NODE_ID
- SESSION_ID
- USER
- CURRENT_DATABASE
- TEIID_SESSION_GET
- TEIID_SESSION_SET
- GENERATED_KEY

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 Teiid 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 for legacy compatibility, see ENV_VAR and SYS_PROP for more appropriately named functions.

ENV(key)

To prevent untrusted access to system properties, this function is not enabled by default. Use the CLI:

```
/subsystem=teiid:write-attribute(name=allow-env-function,value=true)
```

or edit the standalone-teiid.xml file and add following to the "teiid" subsystem

```
<allow-env-function>true</allow-env-function>
```

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)

To prevent untrusted access to environment variables, this function is not enabled by default. Use the CLI:

```
/subsystem=teiid:write-attribute(name=allow-env-function,value=true)
```

or edit the standalone-teiid.xml file and add following to the "teiid" subsystem

```
<allow-env-function>true</allow-env-function>
```

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)

To prevent untrusted access to environment variables, this function is not enabled by default. Use the CLI:

```
/subsystem=teiid:write-attribute(name=allow-env-function,value=true)
```

or edit the standalone-teiid.xml file and add following to the "teiid" subsystem

```
<allow-env-function>true</allow-env-function>
```

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 Teiid 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 previous statement.

GENERATED_KEY(column_name)

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

Null is returned if there is no such generated key nor matching key column. Typically this function will only be used within the scope of procedure to determine a generated key value from an insert. It should not be expected that all inserts provide generated keys as not all sources support returning generated keys.

XML Functions

XML functions provide functionality for working with XML data. See also the JSONTOXML function.

Table of Contents

- Sample Data For Examples
- XMLCAST
- XMLCOMMENT
- XMLCONCAT
- XMLELEMENT
- XMLFOREST
- XMLAGG
- XMLPARSE
- XMLPI
- XMLQUERY
- XMLEXISTS
- XMLSERIALIZE
- XMLTEXT
- XSLTRANSFORM
- XPATHVALUE
- Examples
  - Generating hierarchical XML from flat data structure

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 as XMLTABLE uses to convert values to the desired runtime type - with the exception that array type targets are not supported with XMLCAST.

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]]*)
```

See XMLELEMENT for the definition of NSP - XMLNAMESPACES

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 XMLFORREST to simplify the declaration of multiple XMLELEMENTS, 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>' W
ELLFORMED);

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]]*
```

See XMLELEMENT for the definition of NSP - XMLNAMESPACES

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.

See also FROM Clause#XMLTABLE

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]]*
```

See XMLELEMENT for the definition of NSP - XMLNAMESPACES

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/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) XMLDECL
ARATION])
```

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.

```
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`

Examples

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 Teiid 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
```

another useful link of examples can be found here

JSON Functions

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

Table of Contents

- Sample Data For Examples
- JSONTOXML
- JSONARRAY
- JSONOBJECT
- JSONPARSE
- JSONARRAY_AGG
- Conversion to JSON

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

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.

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.

JSONARRAY

Returns a JSON array.

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

value is any object convertable to a JSON value. 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 convertable to a JSON value. 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 Mar
kets"}]
```

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.

Security Functions

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

Table of Contents

- HASROLE
- MD5
- SHA1
- SHA2_256
- SHA2_512
- AES_ENCRYPT
- AES_DECRYPT

HASROLE

Whether the current caller has the Teiid 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 Teiid Data Roles. 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 a BinaryType data that need to encrypt, 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 a BinaryType data that need to decrypt, the argument `key` is a BinaryType used in decryption.

Spatial Functions

Spatial functions provide functionality for working with geospatial data. Teiid relies on the JTS Topology Suite to provide partial support for the OpenGIS Simple Features Specification For SQL Revision 1.1. Please refer to the specification or to PostGIS for more details about particular functions.

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

Note	Geometry support is still evolving. There may be minor differences between Teiid and pushdown results that will need to be further refined.

Table of Contents

- Conversion Functions
  - ST_GeomFromText
  - ST_GeogFromText
  - ST_GeomFromWKB/ST_GeomFromBinary
  - ST_GeomFromEWKB
  - ST_GeogFromWKB
  - ST_GeomFromText
  - ST_GeomFromGeoJSON
  - ST_GeomFromGML
  - ST_AsText
  - ST_AsBinary
  - ST_AsEWKB
  - ST_AsGeoJSON
  - ST_AsGML
  - ST_AsEWKT
  - ST_AsKML
- Operators
  - &&
- Relationship Functions
  - ST_Contains
  - ST_Crosses
  - ST_Disjoint
  - ST_Distance
  - ST_DWithin
  - ST_Equals
  - ST_Intersects
  - ST_OrderingEquals
  - ST_Overlaps
  - ST_Relate
  - ST_Touches
  - ST_Within
- Attributes and Tests
  - ST_Area
  - ST_CoordDim
  - ST_Dimension
  - ST_EndPoint
  - ST_ExteriorRing
  - ST_GeometryN

    - ST_GeometryType
    - ST_HasArc
    - ST_InteriorRingN
    - ST_IsClosed
    - ST_IsEmpty
    - ST_IsRing
    - ST_IsSimple
    - ST_IsValid
    - ST_Length
    - ST_NumGeometries
    - ST_NumInteriorRings
    - ST_NunPoints
    - ST_PointOnSurface
    - ST_Perimeter
    - ST_PointN
    - ST_SRID
    - ST_SetSRID
    - ST_StartPoint
    - ST_X
    - ST_Y
    - ST_Z
- Misc. Functions
    - ST_Boundary
    - ST_Buffer
    - ST_Centroid
    - ST_ConvexHull
    - ST_CurveToLine
    - ST_Difference
    - ST_Envelope
    - ST_Force_2D
    - ST_Intersection
    - ST_Simplify
    - ST_SimplifyPreserveTopology
    - ST_SnapToGrid
    - ST_SymDifference
    - ST_Transform
    - ST_Union
- Aggregate Functions
    - ST_Extent
- Construction Functions
    - ST_Point
    - ST_Polygon

Conversion Functions

ST_GeomFromText

Returns a geometry from a Clob in WKT format.

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

text is a clob, srid is an optional integer. 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. Only 2 dimensions are supported.

ST_GeogFromWKB

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

```
ST_GeomFromEWKB(bin)
```

bin is a blob. Return value is a geography. Only 2 dimensions are supported.

ST_GeomFromText

Returns a geometry from a Clob in EWKT format.

```
ST_GeomFromEWKT(text)
```

text is a clob. Return value is a geometry. Only 2 dimensions are supported.

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 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.

Operators

&&

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

```
geom1 && geom2
```

geom1, geom2 are geometries. Return value is a boolean.

Relationship Functions

ST_Contains

Returns true if geom1 contains geom2 contains another.

```
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 may 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.

Attributes and Tests

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

Test if the geometry has a circular string. Will currently only report false as curved geometry types are not supported.

```
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 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 geom. 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 guarenteed 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 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. Will typically return null as 3 dimensions are not fully supported.

```
ST_Z(geom)
```

geom is a geometry. Return value is a double.

Misc. Functions

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 geom.

```
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 Teiid.

```
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 geometry. 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 geometry. 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 geometry. Return value is a geometry.

Aggregate Functions

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.

Construction Functions

ST_Point

Retuns the Point for the given cooridinates.

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

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

ST_Polygon

Retuns 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.

Miscellaneous Functions

Documents additional functions and those contributed by other projects.

Table of Contents

- Array functions
  - array_get
  - array_length
- Other Functions
  - uuid
- Data Quality Functions
  - osdq.random
  - osdq.digit
  - osdq.whitespaceIndex
  - osdq.validCreditCard
  - osdq.validSSN
  - osdq.validPhone
  - osdq.validEmail
  - osdq.cosineDistance
  - osdq.jaccardDistance
  - osdq.jaroWinklerDistance
  - osdq.levenshteinDistance
  - osdq.intersectionFuzzy
  - osdq.minusFuzzy
  - osdq.unionFuzzy

Array functions

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 object.

1-based indexing is used. The actual array value should be a java.sql.Array or java array type. A null will be 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 will be thrown if the array value is the wrong type.

Other Functions

uuid

Retuns 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 may 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 that need to randomize.

osdq.digit

Returns digit characters of the string. For example, `a1 b2 c3 d4` will become `1234`

```
digit(sourceValue)
```

The sourceValue is the string that need to digit.

osdq.whitespaceIndex

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

```
whitespaceIndex(sourceValue)
```

The sourceValue is the string that need to find whitespace index.

osdq.validCreditCard

Check whether a Credit Card number is a valid Credit Card number, return `true` if matches credit card logic and checksum.

```
validCreditCard(cc)
```

The cc is the Credit Card number string that need to check.

osdq.validSSN

Check whether a SSN number is a valid SSN number, return `true` if matches ssn logic.

```
validSSN(ssn)
```

The ssn is the SSN number string that need to check.

osdq.validPhone

Check whether a phone number is a valid phone number, return `true` if matches phone logic that more than 8 character less than 12 character, can’t start with 000.

```
validPhone(phone)
```

The phone is the phone number string need to check.

osdq.validEmail

Check whether a email address is a valid email address, return `true` if valid.

```
validEmail(email)
```

The email is the email adress string that need to check.

osdq.cosineDistance

Returns the float distance between two string which base on Cosine Similarity algorithm.

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

The a and b are strings that need to calculate the distance.

osdq.jaccardDistance

Returns the float distance between two string which base on Jaccard similarity algorithm.

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

The a and b are strings that need to calculate the distance.

osdq.jaroWinklerDistance

Returns the float distance between two string which base on Jaro-Winkler algorithm.

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

The a and b are strings that need to calculate the distance.

osdq.levenshteinDistance

Returns the float distance between two string which base on Levenshtein algorithm.

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

The a and b are strings that need 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)
```

The 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)
```

The 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)
```

The 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.

Nondeterministic Function Handling

Teiid 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.

1. **Deterministic** - the function will always return 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 is treated as such for performance. All functions not categorized below are considered deterministic.
2. **User Deterministic** - the function will return 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.
3. **Session Deterministic** - the function will return 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.
4. **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.
5. **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.

DML Commands

Teiid supports SQL for issuing queries and for defining view transformations; see also Procedure Language for how SQL is used in virtual procedures and update procedures. Nearly all these features follow standard SQL syntax and functionality, so any SQL reference can be used for more information.

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

In addition, procedures can be executed using the EXECUTE command, through a Procedural Relational Command, or an Anonymous Procedure Block.

SELECT Command

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

A SELECT command has a number of clauses:

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

All of these clauses other than OPTION are defined by the SQL specification. The specification also specifies the order that these clauses will be logically processed. Below is the processing order where each stage passes a set of rows to the following stage. Note that this processing model is logical and does not represent the way any actual database engine performs the processing, although it is a useful model for understanding questions about SQL.

- WITH stage - gathers all rows from all with items in the order listed. Subsequent with items and the main query can reference the a with item as if it is a table.
- FROM stage - gathers all rows from all tables involved in the query and logically joins them with a Cartesian product, producing 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.
- WHERE stage - applies a criteria to every output row from the FROM stage, further reducing the number of rows.
- GROUP BY stage - groups sets of rows with matching values in the group by columns.
- HAVING stage - 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).

- SELECT stage - specifies the column expressions that should be returned from the query. Expressions are evaluated, including aggregate functions 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 will be performed on the rows being returned from the SELECT stage.
- ORDER BY stage - 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.
- LIMIT stage - returns only the specified rows (with skip and limit values).

This model can be used to understand many questions about SQL. For example, columns aliased in the SELECT clause can only be referenced by alias in the ORDER BY clause. 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`.

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 - "SELECT value, …. UNION ALL SELECT valueX, …".

Update Commands

Update commands can report integer update counts. If a larger number or rows is updated, then the max integer value will be reported (2^31 -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
```

UPDATE Command

The UPDATE command is used to modify records in a table. The operation may result 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]
```

DELETE Command

The DELETE command is used to remove records from a table. The operation may result 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]
```

UPSERT/MERGE Command

The UPSERT (or MERGE) is used to add and/or update records. The Teiid specific (non-ANSI) UPSERT is simply a modified INSERT statement that requires the target table to have a primary key and for the target columns to cover the primary key. The UPSERT operation will then check the existence of each row prior to INSERT and instead perform an UPDATE if the row already exists.

Example Syntax

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

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

Note	**UPSERT Pushdown** - If UPSERT statement is not pushed to the source, it will be broken down into the respective insert/update operations, which requires XA support on the target system to guarantee atomicity.

EXECUTE Command

The EXECUTE command is used to execute a procedure, such as a virtual procedure or a stored procedure. Procedures may 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. Note that EXEC or CALL can be used as a short form of this command.

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 are have default values and/or 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 are have default values and/or 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 will be 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.

Procedural Relational Command

Procedural relational commands use the syntax of a SELECT to emulate an EXEC. In a procedural relational command a procedure group names is used in a FROM clause in place of a table. That procedure will be 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 found in the criteria results in an 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.c
ol2
```

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 will be projected as two columns, one that represents the output value and one named {column name}_IN that represents the input of the parameter.
- Input values are passed via criteria. Values can be passed by '=','is null', or '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 which allows it to be planned correctly as a dependent join when necessary or fail when sufficient criteria cannot be found.
- Procedures containing duplicate names between the parameters (IN, IN_OUT, OUT, RETURN) and result set columns cannot be used in a procedural relational command.
- Default values for IN, 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.

Multiple Execution

The usage of 'in' or join criteria can result in the procedure being executed multiple times.

Alternative Syntax

None of issues listed in the syntax rules above exist if a nested table reference is used.

Anonymous Procedure Block

A Procedure Language block may be executed as a user command. This is advantageous in situations when a virtual procedure doesn’t exists, but a set of processing can be cared out on the server side together.

Example Syntax

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

Syntax Rules:

- In parameters are supported with prepared/callable statement parameters as shown above with a ? parameter.
- out parameters are not yet supported - consider using session variables as a workaround as needed.
- a return parameter is not supported.
- a single result will be returned if any of the statements returns a result set. All returnable result sets must have a matching number of columns and types. Use the WITHOUT RETURN clause to indicate that a statement is not intended to a result set as needed.

Set Operations

Teiid supports the UNION, UNION ALL, INTERSECT, EXCEPT set operation as a way of combining 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 will be 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.
- INTERSECT ALL, and EXCEPT ALL are currently not supported.

Subqueries

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

Supported 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.

Subqueries are supported 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.

WITH Clause

Teiid supports non-recursive common table expressions via the WITH clause. WITH clause items may be referenced as tables in subsequent with clause items and in the main query. The WITH clause can be thought of 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 Teiid. This forces a single evaluation of the common table.

Note	The WITH clause is also subject to optimization and it’s entries may 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 will likely be inlined. In some situations, such as when using a common table to prevent n-many processing of a non-pushdown correlated subquery, you may 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. Processing flows with The anchor query expression executed first. The results will be added to the common table and will be referenced for the execution of the recursive query expression. The process will be repeated against the new results until there are no more intermediate results.

Note	A non terminating recursive common table expression can lead to excessive processing.

To prevent runaway processing of a recursive common table expression, processing is by default limited to 10000 iterations. Recursive common table expressions that are pushed down are not subject to this limit, but may be subject to other source specific limits. The limit can be modified by setting the session variable teiid.maxRecusion to a larger integer value. Once the max has been exceeded an exception will be thrown.

Example:

```
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
```

This will fail to process as the recursion limit will be reached before processing completes.

SELECT Clause

SQL queries that start with the SELECT keyword and are often referred to as "SELECT statements". Teiid supports most of the standard SQL query constructs.

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.

FROM Clause

The FROM clause specifies the target table(s) 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
- 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
- FROM TEXTTABLE…
- FROM XMLTABLE…
- FROM ARRAYTABLE…
- FROM OBJECTTABLE…
- FROM (SELECT …

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 Joins

MAKEIND, MAKEDEP, and MAKENOTDEP are hints used to control dependent join behavior. They should only be used in situations where the optimizer does not choose the most optimal plan based upon query structure, metadata, and costing information. The hints may appear in a comment that proceeds the from clause. The hints can be specified against any from clause, not just a named table.

- MAKEIND - treat this clause as the independent (feeder) side of a dependent join if possible.
- MAKEDEP - treat this clause as the dependent (filtered) side of a dependent join if possible.

- MAKENOTDEP - do not treat this clause as the dependent (filtered) side of a join.

MAKEDEP and MAKEIND support optional max and join arguments:

- MAKEDEP(JOIN) means that the entire join should be pushed
- MAKEDEP(NO JOIN) means that the entire join should not be pushed
- MAKEDEP(MAX:val) meaning that the dependent join should only be performed if there are less than the max 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 merge the nested SQL in the surrounding query - also known as view flattening. This hint only applies to Teiid planning and is not passed to source queries. NO_UNNEST may appear in a comment that proceeds the from clause.

The PRESERVE hint can be used against an ANSI join tree to preserve the structure of the join rather than allowing the Teiid 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)
```

Nested Table Reference

Nested tables may appear in the FROM clause with the TABLE keyword. They are an alternative to using a view with normal join semantics. The columns projected from the command contained in the nested table may be used just as any of the other FROM clause projected columns in join criteria, the where clause, etc.

A nested table may 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 example:

```
select * from t1, TABLE(call proc(t1.x)) t2
```

Invalid example, since t1 appears after the nested table in the from clause:

```
select * from TABLE(call proc(t1.x)) t2, t1
```

Note	**Multiple Execution** - The usage of a correlated nested table may result in multiple executions of the table expression - once for each correlated row.

XMLTABLE

The XMLTABLE function uses XQuery to produce tabular output. The XMLTABLE function is implicitly a nested table and may be correlated to preceding FROM clause entries. 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]))
```

See XMLELEMENT for the definition of NSP - XMLNAMESPACES.

See XMLQUERY for the definition of PASSING.

See also XMLQUERY

Note	See also XQuery Optimization

Parameters

- The optional XMLNAMESPACES clause specifies the namepaces for use in the XQuery and COLUMN path expressions.
- The xquery-expression should be a valid XQuery. Each sequence item returned by the xquery will be used to create a row of values as defined by the COLUMNS clause.
- If COLUMNS is not specified, then that is the same as having the COLUMNS clause: "COLUMNS OBJECT_VALUE XML PATH '.'", which returns the entire item as an XML value.
- A FOR ORDINALITY column is typed as integer and will return the 1-based item number as its value.
- Each non-ordinality column specifies a type and optionally a PATH and a DEFAULT expression.
- If PATH is not specified, then the path will be the same as the column name.

Syntax Rules:

- Only 1 FOR ORDINALITY column may be specified.
- The columns names must not contain duplicates.
- The blob datatype is supported, but there is only built-in support for xs:hexBinary values. For xs:base64Binary, use a workaround of a PATH that uses the 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 will be 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 as it is effectively the empty string. The xsi:nil attribute should be used to define convey a null valued element.

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. Without the nil attribute an exception would be thrown converting b to an integer value.

```
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
```

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 may be correlated to preceding FROM clause entries.

Usage:

```
ARRAYTABLE(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.

Syntax Rules:

- The columns names must be not contain duplicates.

Examples

- As a nested table:

```
select x.* from (call source.invokeMDX('some query')) r, arraytable(r.tuple COLUMNS first string, second bigdec
imal) x
```

ARRAYTABLE is effectively a shortcut for using the Miscellaneous Functions#array_get function in a nested table. For example

```
ARRAYTABLE(val COLUMNS col1 string, col2 integer) AS X
```

is the same as

```
TABLE(SELECT cast(array_get(val, 1) AS string) AS col1, cast(array_get(val, 2) AS integer) AS col2) AS X
```

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 may be correlated to preceding FROM clause entries.

Usage:

```
OBJECTTABLE([LANGUAGE lang] rowScript [PASSING val AS name ...] COLUMNS colName colType colScript [DEFAULT defa
ultExpr] ...) 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. In some instances this may mean making additional modules available to your vdb, which can be done via the same process as adding modules/libraries for UDFs. If a LANGUAGE is not specified, the default of 'teiid_script' (see below) will be used.
- name - an identifier that will bind the val expression value into the script context.
- rowScript is a string literal specifying the script to create the row values. for each non-null item the Iterator produces the columns will be evaluated.
- colName/colType are the id/data type of the column, which can optionally be defaulted with the DEFAULT clause expression defaultExpr.
- colScript is a string literal specifying the script that evaluates to the column value.

Syntax Rules:

- The columns names must be not contain duplicates.
- Teiid will place 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 will be obtained. Any other Object will be treated as an Iterator of a single item). In all cases null row values will be skipped.

Note	While there is no restriction what can be used as a PASSING variable names you should choose names that can be referenced as identifiers in the target language.

Examples

- Accessing special variables:

```
SELECT x.* FROM OBJECTTABLE('teiid_context' COLUMNS "user" string 'teiid_row.userName', row_number integer 'tei
id_row_number') AS x
```

The result would be a row with two columns containing the user name and 1 respectively.

Note	Due to their mostly unrestricted access to Java functionality, usage of languages other than teiid_script is restricted by default. A VDB must declare all allowable languages by name in the allowed-languages VDB Properties using a comma separated list. The names are case sensitive names and should be separated without whitespace. Without this property it is not possible to use OBJECTTABLE even from within view definitions that are not subject to normal permission checks.

Data roles can define the language permission.

teiid_script

teiid_script is a simple scripting expression language that allows access to passing and special variables as well as any 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 may 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 both a `getFoo()` and `foo()` method, then the accessor foo references `fo ()` and getFoo should be used to call the getter. An array or list index may be accessed using a 1-based positive integral value - using the same '.' accessor syntax. The same logic as the system function array_get is used meaning that null will be returned rather than exception if the index is out of bounds.

teiid_script is effectively dynamically typed as typing is performed at runtime. If a accessor does not exist on the object or if the method is not accessible, then an exception will be raised. If at any point in the accessor chain evaluates to a null value, then null will be returned.

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
```

TEXTTABLE

The TEXTTABLE function processes character input to produce tabular output. It supports both fixed and delimited file format parsing. The function itself defines what columns it projects. The TEXTTABLE function is implicitly a nested table and may be correlated to preceding FROM clause entries.

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 CLOB.
- SELECTOR is 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 indicates that fixed parsing should not assume the presence of newline row delimiters.
- ROW DELIMITER sets the row delimiter / new line to an alternate character. Defaults to the new line character - with built in handling for treating carriage return new line as a single character. If ROW DELIMITER is specified, carriage return will be 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 may still be specified with SKIP.
- A FOR ORDINALITY column is typed as integer and will return the 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 specified on the TEXTTABLE, it will affect all column and header values. If NO TRIM is specified on a column, then the fixed or unqualified text value not be 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 ESCAPE, QUOTE, column SELECTOR, nor HEADER should not be specified.
- If width is not specified, then NO ROW DELIMITER cannot be used.
- The columns names must not contain duplicates.
- The QUOTE, DELIMITER, and ROW DELIMITER must all be different characters.

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 widt
h 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
```

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

GROUP BY Clause

The GROUP BY clause denotes that rows should be grouped according to the specified expression values. One row will be 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. For example with the normal aggregation query

```
SELECT country, city, sum(amount) from sales group by country, city
```

returning:

country	city	sum(amount)
US	St. Louis	10000
US	Raleigh	150000
US	Denver	20000
UK	Birmingham	50000
UK	London	75000

The 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 support ROLLUPs and some optimizations compared to normal aggregate processing may be inhibited by the use of a ROLLUP.

Teiid’s support for ROLLUP is more limited than the SQL specification. In future releases support for CUBE, grouping sets, and more than a single extended grouping element may be supported.

HAVING Clause

The HAVING clause operates exactly as a WHERE clause although it operates on the output of a GROUP BY. It supports the same syntax as the WHERE clause.

Syntax Rules:

- Expressions used in the group by clause must either contain an aggregate function: COUNT, AVG, SUM, MIN, MAX. or be one of the grouping expressions.

ORDER BY Clause

The ORDER BY clause specifies how records should be sorted. The options are ASC (ascending) and DESC (descending).

Usage:

```
ORDER BY expression [ASC|DESC] [NULLS (FIRST|LAST)], ...
```

Syntax Rules:

- Sort columns may 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 may appear in the SELECT clause as the expression for an aliased column or may reference columns from tables in the FROM clause. If the column reference is not in the SELECT clause the query must not 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 will be removed by the Teiid 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 Teiid’s internal default sorting behavior. However not all sources return results with nulss sorted as low values by default, and Teiid may 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 Teiid. It is preferable to use alias names in the order by clause.

LIMIT Clause

The LIMIT clause specifies a limit on the number of records returned from the SELECT command. An optional offset (the number of rows to skip) can be specified. 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 will return no rows.
- The terms FIRST/NEXT are interchangeable as well as ROW/ROWS.
- The limit clause may 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, e.g. "SELECT * FROM VW /*+ NON_STRICT */ LIMIT 2".

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

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.

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.
- Teiid’s support for SELECT INTO is similar to MS SQL Server. The target of the INTO clause is a table where the result of the rest select command will be inserted. SELECT INTO should not be used UNION query.

OPTION Clause

The OPTION keyword denotes options the user can pass in with the command. These options are Teiid specific and 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 dependent 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 name may match either an alias name or the fully qualified 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) meaning that the dependent join should only be performed if there are less than the max number of values from the independent side.

Tip	Previous versions of Teiid accepted the PLANONLY, DEBUG, and SHOWPLAN option arguments. These are no longer accepted in the OPTION clause. Please see the Client Developers Guide for replacements to those options.

Note	MAKEDEP and MAKENOTDEP hints may 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.

DDL Commands

Teiid supports a subset of DDL at runtime to create/drop temporary tables and to manipulate procedure and view definitions. It is not currently possible to arbitrarily drop/create non-temporary metadata entries. See DDL Metadata for DDL used within a VDB to define schemas.

Note	A `MetadataRepository` must be configured to make a non-temporary metadata update persistent. See the Developers Guide Runtime Metadata Updates section for more.

Temp Tables

Teiid supports creating temporary(or "temp") tables. Temp tables are dynamically created, but are treated as any other physical table.

Table of Contents

- Local Temporary Tables
  - Example
- Global Temporary Tables
- Global and Local Temporary Table Features
- Foreign Temporary Tables

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 `#`.

Explicit Creation syntax

```
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

```
INSERT INTO #name (column, ...) VALUES (value, ...)
INSERT INTO #name [(column, ...)] select c1, c2 from t
```

If #x doesn't exist, it will be defined using the given column names and types from the value expressions, or the target column names (in not supplied, the column names will match the derived column names from the query), and the types from the query derived columns.

Note	Teiid's interpretation of local is different than the SQL specification and other database vendors. Local means that the scope of temp table will be either to the session or the block of a virtual procedure that creates it. Upon exiting the block or the termination of the session the table is dropped. Session and any other temporary tables created in calling procedures are not visible to called procedures. If a temporary table of the same name is created in a called procedure a new instance is created.

Drop syntax

```
DROP TABLE name
```

Example

The following example is a series of statements that loads a temporary table with data from 2 sources, and with a manually inserted record, and then uses that temp table in a subsequent query.

```
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;
```

See Virtual Procedures for more on local temporary table usage.

Global Temporary Tables

Global temporary tables are created via the metadata supplied to Teiid at deploy time. Unlike local temporary tables, they cannot be created at runtime. A global temporary tables share a common definition via a schema entry, but each session has a new instance of the temporary table created upon it's first use. 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 the SERIAL data type is used, then each session's instance of the global temporary table will use it's own sequence.

See the CREATE TABLE DDL statement for all syntax options.

Currently UPDATABLE must be explicitly specified for the temporary table to be updated.

Global and Local Temporary Table Features

Primary Key Support:

- All key columns must be comparable.
- Use of a primary key creates a clustered index that supports search improvements for comparison, in, like, and order by.
- Null is an allowable primary key value, but there must be only 1 row that has an all null key.

Transaction Support:

- Temp tables support a READ_UNCOMMITED transaction isolation level. There are no locking mechanisms available to support 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 1 transaction at a time. This mode of operation is ensured by connection pooling that tracks connections by transaction.

Limitations:

- With the CREATE TABLE syntax only basic table definition (column name, type, and nullable information) and an optional primary key are supported. For global temporary tables additional metadata in the create statement is effectively ignored when creating the temporary table instance - but may still be utilized by planning similar to any other table entry.
- Similar to PostgreSQL, Teiid defaults to ON COMMIT PRESERVE ROWS. No other ON COMMIT action is supported at this time.
- The "drop behavior" option is not supported in the drop statement.
- Temp tables are not fail-over safe.
- Non-inlined lob values (xml, clob, blob) are tracked by reference rather than by value in a temporary table. Lob values from external sources that are inserted in a temporary table may become unreadable when the associated statement or connection is closed.

Foreign Temporary Tables

Unlike Teiid local or global temporary tables, a foreign temporary table is a reference to a 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. In general, usage of DDL OPTION clauses may be required to properly access the source table, including setting the name in source, updatability, native types, etc.

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 Teiid 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 Teiid schema and will be scoped to the session or procedure block where created.

The DROP syntax for a foreign temporary table is the same as for a non-foreign temporary table.

Note	Neither a CREATE nor a corresponding DROP of a foreign temporary table issue a pushdown command, rather this mechanism simply exposes a source table for use within Teiid 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 Teiid 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 Teiid
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. Teiid does not currently attempt to pushdown a source creation of a temporary table based upon 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 support session scoped temporary tables (such as PostgreSQL) will work if accessed under a transaction. A transaction is necessary because:

- the source on commit behavior (most likely DELETE ROWS or DROP) will ensure clean-up. Keep in mind that a Teiid drop does not issue a source command and is not guaranteed to occur (in some exception cases, loss of db connectivity, hard shutdown, etc.).
- the source pool when using track connections by transaction will ensure that multiple uses of that source by Teiid will use the same connection/session and thus the same temporary table and data.

Tip	Since Teiid does not yet support the ON COMMIT clause it's important to consider that the source table ON COMMIT behavior will likely be different that the default, PRESERVE ROWS, for Teiid local temporary tables.

Alter View

Usage:

```
ALTER VIEW name AS queryExpression
```

Syntax Rules:

- The alter query expression may be prefixed with a cache hint for materialized view definitions. The hint will take effect the next time the materialized view table is loaded.

Alter Procedure

Usage:

```
ALTER PROCEDURE name AS block
```

Syntax Rules:

- The alter block should not include `CREATE VIRTUAL PROCEDURE`
- The alter block may be prefixed with a cache hint for cached procedures.

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 yet true schema objects. They are scoped only to their view and have no name.
- An Update Procedures (Triggers) must already exist for the given trigger event.

Procedures

Teiid supports calling foreign procedures and defining virtual procedures and triggers using a procedure language.

Procedure Language

Teiid supports a procedural language for defining 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 (Triggers).

Table of Contents

- Command Statement
- Dynamic SQL Command
- Declaration Statement
- Assignment Statement
- Special Variables
- Compound Statement
  - Exception Handling
- If Statement
- Loop Statement
- While Statement
- Continue Statement
- Break Statement
- Leave Statement
- Return Statement
- Error Statement
- Raise Statement
- Exception Expression

Command Statement

A command statement executes a DML Command, such as SELECT, INSERT, UPDATE, DELETE, EXECUTE, or a DDL statement, dynamic SQL, etc.

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 parameter will be implicitly convertable to the datatype of the variable.
- 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.

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 <variab
le>=<expression> [,<variable>=<expression>]*] [UPDATE <literal>]
```

Syntax Rules:

- The sql expression must be a clob/string value 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 legal expressions. It is not possible to use the "USING" clause to replace table names, keywords, etc. 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.

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 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 US
ING ID=AccountAccess.GetAccounts.ID, LastName=AccountAccess.GetAccounts.LastName, BirthDay=AccountAccess.GetAcc
ounts.Bday;
...
```

Known Limitations and Work-Arounds

The use of dynamic SQL command results in an assignment statement requires the use of a temp 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))';
```

The preferred approach is for the user to ensure the criteria is not NULL prior its usage. If this is not possible, a good approach is to 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 with AS and INTO clauses

```
/* Execute an update */
EXECUTE IMMEDIATE <expression>;
```

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.

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 as their fully qualified name.

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
```

Special Variables

VARIABLES.ROWCOUNT integer variable will contain the numbers of rows affected by the last insert/update/delete command statement executed. Inserts that are processed by dynamic sql with an into clause will also update the *ROWCOUNT*.

Usage:

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.

Compound Statement

A compound statement or block logically groups a series of statements. Temporary tables and variables created in a compound statement are local only to that block are destroyed when exiting the block.

Usage:

```
[label :] BEGIN [[NOT] ATOMIC]
    statement*
[EXCEPTION ex
    statement*
]
END
```

Note	When a block is expected by a IF, LOOP, WHILE, etc. a single statement is also accepted by the parser. Even though the block BEGIN/END are not expected, the statement will execute as if wrapped in a BEGIN/END pair.

Syntax Rules

- IF NOT ATOMIC or no ATOMIC clause is specified, the block will be executed non-atomically.
- IF ATOMIC the block must execute atomically. If a transaction is already associated with the thread, no additional action will be taken - savepoints and/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 other label used in statements containing 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 the EXCEPTION clause is used with in 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 Teiid 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 exception group will contain:

Variable	Type	Description
STATE	string	The SQL State
ERRORCODE	integer	The error or vendor code. In the case of Teiid internal exceptions this will be the integer suffix of the TEIIDxxxx code
TEIIDCODE	string	The full Teiid 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	Teiid does not yet fully comply with the ANSI SQL specification on SQL State usage. For Teiid errors without an underlying SQLException cause, it is best to use the Teiid code.

The exception group name may 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
```

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 may be any valid boolean expression or an IS DISTINCT FROM predicate referencing row values. This IS DISTINCT FROM extension uses the 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.

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 other label used in statements containing this one.

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 other label used in statements containing this one.

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.

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.

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.

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 convertable to the expected type.
- Even if the procedure has a return value, it is not required to specify a return value in a RETURN statement.

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;
```

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';
```

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 specifying the exception message and SQL state respectively. Teiid does not yet 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 specifying 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.

Virtual Procedures

Virtual procedures are defined using the Teiid procedural language. A virtual procedure has zero or more input/inout/out parameters, an optional return parameter, and an optional result set. Virtual procedures support the ability to execute queries and other SQL commands, define temporary tables, add data to temporary tables, walk through result sets, use loops, and use conditional logic.

Table of Contents

- Virtual Procedure Definition
- Procedure Parameters
- Example Virtual Procedures
- Executing Virtual Procedures
- Limitations

Virtual Procedure Definition

In DDL: DDL Metadata#Create Procedure/Function

Note that the optional result parameter is always considered the first parameter

Within the body of the procedure, any valid statement may be used.

There is no explicit cursoring or value returning statement, rather 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.

Procedure Parameters

Virtual procedures can take zero or more IN/INOUT parameters and may also 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 the fully-qualified name of the param (or less if unambiguous). For example, MySchema.MyProc.Param1.

Example of 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 remain the value it was assigned for input. Any OUT/RETURN parameter not assigned a value will remain the as the default NULL value. The INOUT/OUT/RETURN output values are validated against the NOT NULL metadata of the parameter.

Example Virtual Procedures

This example is 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
```

This example is 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 PartsVi
rtual.SupplierInfo.PART_ID;
  END ELSE
  BEGIN
    SELECT * FROM PartsVirtual.SupplierInfo WHERE QUANTITY > PartsVirtual.OrderedQtyProc.QTYIN ORDER BY PartsVi
rtual.SupplierInfo.PART_ID DESC;
  END
END
```

Executing Virtual Procedures

You execute procedures using the SQL EXECUTE command. 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 will return a result set just 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
```

Limitations

Teiid virtual procedures may only return 1 result set. If you need to pass in a result set or pass out multiple result set, then consider using global temporary tables.

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. Teiid can perform update operations against views. Update commands - INSERT, UPDATE, or DELETE - against a view 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 a user's update command against a view should be 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.

Teiid supports INSTEAD OF triggers on views similar to traditional databases. There may only be 1 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 view this SQL command is executed against is detected.
3. The correct procedure is chosen depending upon whether the command is an INSERT, UPDATE, or DELETE.
4. The procedure is executed. The procedure itself can contain SQL commands of its own which can be of different types than the command submitted by the user application that invoked the procedure.
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

Teiid supports AFTER triggers on source tables, which are called by events from a CDC (change data capture) system.

Usage:

```
CREATE TRIGGER ON source_table AFTER INSERT|UPDATE|DELETE AS
FOR EACH ROW
...
```

For Each Row

Only the FOR EACH ROW construct is supported as a trigger handler. A FOR EACH ROW trigger procedure will evaluate its block for each row of the view/source affected by the associated change. For UPDATE and DELETE statements this will be every row that passes the WHERE condition. For INSERT statements there will be 1 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.

Definition

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.

Tip	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

You can use a number of special variables when defining your update procedure.

NEW Variables

Every attribute on 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 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 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 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 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 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

Comments

Teiid supports multi-line comments enclosed with /* */:

```
/* comment
comment
comment... */
```

And single line comments:

```
SELECT ... -- comment
```

Comment nesting is supported.

Datatypes

The Teiid type system is based upon Java/JDBC types. The runtime object will be represented by the corresponding Java class, such as Long, Integer, Boolean, String, etc.

The type system can be extended using domain types.

Supported Types

Teiid supports a core set of runtime types. Runtime types can be different than semantic types 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.

Table 1. **Teiid 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 Unicode character	java.lang.Character	CHAR	CHAR
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
date	datetime, representing a single day (year, month, day)	java.sql.Date	DATE	DATE
time	datetime, representing a single time (hours, minutes, seconds, milliseconds)	java.sql.Time	TIME	TIME
timestamp	datetime, representing a single date and time (year, month, day, hours, minutes, seconds, milliseconds, nanoseconds)	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
json (11.2+)	character large object, representing a stream of json characters	java.sql.Clob [7]	CLOB	VARCHAR

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. The geospatial types act in a similar manner. Extension metadata may be needed for srid, type, and number of dimensions for consumption by tools/OData - but it is not yet enforced. In some instances you may need to use the ST_SETSRID function to ensure the srid is associated.

Reference Link

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

Arrays

Warning	Teiid’s support for arrays is a new feature as of the 8.5 release. Support will be refined and enhanced in subsequent releases.

An array of any type is designated by adding [] for each array dimension to the type declaration.

Example array types:

```
string[]
```

```
integer[][]
```

Note	Teiid array handling is typically in memory. It is not advisable to rely on the usage of large array values. Also arrays of lobs are not well supported and will typically not be handled correctly when serialized.

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 may be explicitly converted to any other type.
- The NULL value may be converted to any type.
- Any valid implicit conversion is also a valid explicit conversion.
- Situations involving literal values that would normally require explicit conversions may have the explicit conversion applied implicitly if no loss of information occurs.
- If widenComparisonToString is false (the default), when Teiid detects that an explicit conversion can not be applied implicitly in criteria, then an exception will be raised. If widenComparisonToString is true, then depending upon the comparison a widening conversion will be applied or the criteria will be treated as false.

For example:

```
SELECT * FROM my.table WHERE created_by = 'not a date'
```

With widenComparisonToString as false and created_by is typed as date, rather than converting `not a date` to a date value, an exception will be raised.

- Explicit conversions that are not allowed between two types will result in an exception before execution. Allowed explicit conversions may still fail during processing if the runtime values are not actually convertable.

Warning	The Teiid conversions of float/double/bigdecimal/timestamp to string rely on the JDBC/Java defined output formats. Pushdown behavior attempts to mimic these results, but may vary depending upon the actual source type and conversion logic. Care should be taken to not assume the string form in criteria or other places where a variation may cause different results.

Table 1. **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 [1]
char	string	
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 [2], double [2]	boolean, byte, short, integer, float, double
biginteger	string, bigdecimal float [2], double [2]	boolean, byte, short, integer, long, float, double
bigdecimal	string, float [2], double [2]	boolean, byte, short, integer, long, biginteger, float, double
float	string, bigdecimal, double	boolean, byte, short, integer, long, biginteger
double	string, bigdecimal, float [2]	boolean, byte, short, integer, long, biginteger, float
date	string, timestamp	
time	string, timestamp	
timestamp	string	date, time
clob		string
json	clob	string
xml		string [3]
geography		geometry

1. string to xml is equivalent to XMLPARSE(DOCUMENT exp) - See also XML Functions#XMLPARSE
2. implicit conversion to float/double only occurs for literal values
3. xml to string is equivalent to XMLSERIALIZE(exp AS STRING) - see also XML Functions#XMLSERIALIZE

Special Conversion Cases

Conversion of String Literals

Teiid 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:

```
SELECT * FROM my.table WHERE created_by = '2016-01-02'
```

Here if the created_by column has the datatype of date, Teiid automatically converts the string literal to a date datatype as well.

Converting to Boolean

Teiid can automatically convert literal strings and numeric type values to Boolean values as follows:

Type	Literal Value	Boolean Value
String	'false'	false
	'unknown'	null
	other	true
Numeric	0	false
	other	true

Date/Time/Timestamp Type Conversions

Teiid can implicitly convert properly formatted literal strings to their associated date-related datatypes as follows:

String Literal Format	Possible Implicit Conversion Type
yyyy-mm-dd	DATE
hh:mm:ss	TIME
yyyy-mm-dd[hh:mm:ss.[fff…]]	TIMESTAMP

The formats above are those expected by the JDBC date types. To use other formats see the functions `PARSEDATE` , `PARSETIME` , `PARSETIMESTAMP` .

Escaped Literal Syntax

In addition to standard SQL syntax, datatype values may be expressed directly in SQL using escape syntax to define the type. Note that the supplied string value 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…]]'

Updatable Views

Any view may be marked as updatable. In many circumstances the view definition may allow 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 is itself inherently updatable. A view defined by a UNION ALL can support inherent INSERTs if it is a Federated Optimizations#Partitioned Union and the INSERT specifies values that belong to a single partition.

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 may 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, it is required:

- An INSERT/UPDATE can only modify a single Key-preserved Table.
- To allow DELETE operations there must be only a single Key-preserved Table.

If the default handling is not available or you wish to have an alternative implementation of an INSERT/UPDATE/DELETE, then you may use Update Procedures (Triggers) to define procedures to handle the respective operations.

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 as it targets a single key-preserved table - child_table. However "insert into denormalized (name) values ('a')" would fail as it maps to a parent_table which is not key preserved as there can be multiple rows for each parent_table key. Also an insert against just parent_table may not be visible to the view - as there may be no child entities associated either.

Not all scenarios will work. Referencing the above 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 as the logic doesn’t yet consider the equivalency of the key values. If you encounter a scenario that needs support, please log an issue.

Key-preserved Table

A key-preserved table has a primary or unique key that would remain unique if it were 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.

Transaction Support

Teiid utilizes XA transactions for participating in global transactions and for demarcating its local and command scoped transactions. JBoss Transactions is used by Teiid as its transaction manager. See this documentation for the advanced features provided by JBoss Transactions.

Table 1. **Teiid 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	Teiid participates in a global transaction as an XA Resource.

The default transaction isolation level for Teiid is READ_COMMITTED.

AutoCommitTxn Execution Property

Since user level commands may execute multiple source commands, users can specify the AutoCommitTxn execution property to control the transactional behavior of a user command when not in a local or global transaction.

Table 1. **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 Teiid based upon command type, the transaction isolation level, and available metadata. A wrapping transaction is not needed if:

- If a user command is fully pushed to the source.
- If the user command is a SELECT (including XML) and the transaction isolation is not REPEATABLE_READ nor SERIALIABLE.
- If the user command is a stored procedure and the transaction isolation is not REPEATABLE_READ nor SERIALIABLE and the Updating Model Count is zero.

The update count may be set on all procedures as part of the procedure metadata in the model.

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 1. **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). Also the success or failure of that update corresponds to the success of 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.

JDBC and Transactions

JDBC API Functionality

The transaction scopes above map to these 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 Teiid 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 three 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 app server container begins and ends a transaction automatically.

In any of these cases, transactions may be either local or XA transactions, depending on how the code and descriptors are written. Some kinds of beans (stateful session beans and entity beans) are not required by the spec to support non-transactional sources, although the spec does allow an app server to optionally support this with the caution that this is not portable or predictable. Generally speaking, to support most typical EJB activities in a portable fashion requires some kind of transaction support.

Transactional Behavior with WildFly Data Source Types

WildFly allows creation of different types of data sources, based on their transactional capabilities. The type of data source you create for your VDB’s sources also dictates if that data source will be participating the distributed transaction or not, irrespective of the transaction scope you selected from above. Here are different types of data sources

- xa-datasource: Capable of participating in the distributed transaction using XA. This is recommended type be used with any Teiid sources.
- local-datasource: Does not participate in XA, unless this is the *only* source that is local-datasource that is participating among other xa-datasources in the current distributed transaction. This technique is called last commit optimization. However, if you have more then one local-datasources participating in a transaction, then the transaction manager will end up with *"Could not enlist in transaction on entering meta-aware object!;"* exception.
- no-tx-datasource: Does not participate in distributed transaction at all. In the scope of Teiid command over multiple sources, you can include this type of datasource in the same distributed transaction context, however this source will be it will not be subject to any transactional participation. Any changes done on this source as part of the transaction scope, can not be rolled back. If you have three different sources A, B, C and they are being used in Teiid. Here are some variations on how they behave with different types of data sources. The suffixes "xa", "local", "no-tx" define different type of sources used.
- A-xa B-xa, C-xa : Can participate in all transactional scopes. No restrictions.
- A-xa, B-xa, c-local: Can participate in all transactional scopes. Note that there is only one single source is "local". It is assumed that in the Global scope, the third party datasource, other than Teiid Datasource is also XA.
- A-xa, B-xa, C-no-tx : Can participate in all transactional scopes. Note "C" is not a really bound by any transactional contract. A and B are the only participents in XA transaction.
- A-xa, B-local, C-no-tx : Can participate in all transactional scopes. Note "C" is not a really bound by any transactional contract, and there is only single "local" source.
- If any two or more sources are "local" : They can only participate in Command mode with "autoCommitTxn=OFF". Otherwise will end with exception as "Could not enlist in transaction on entering meta-aware object!;" exception, as it is not possible to do a XA transaction with "local" datasources.
- A-no-tx, B-no-tx, C-no-tx : Can participate in all transaction scopes, but none of the sources will be bound by transactional terms. This is equivalent to not using transactions or setting Command mode with "autoCommitTxn=OFF".

To create XA data source, look in the WildFly "doc" directory for example templates, or use the "admin-console" to create the XA data sources.

If your datasource is not XA, and not the only local source and can not use "no-tx", then you can look into extending the source to implement the compensating XA implementation. i.e. define your own resource manager for your source and manage the transaction the way you want it to behave. Note that this could be complicated if not impossible if your source natively does not support distributed XA protocol. In summay

- Use XA datasource if possible
- Use no-tx datasource if applicable
- Use autoCommitTxn = OFF, and let go distributed transactions, though not recommended
- Write a compensating XA based implementation.

Table 1. **Teiid Transaction Participation**

Teiid-Tx-Scope	XA source	Local Source	No-Tx Source

Local (Auto-commit=false)	always	Only If Single Source	never
Global	always	Only If Single Source	never
Auto-commit=true, AutoCommitTxn=ON, or DETECT and txn started	always	Only If Single Source	never
Auto-commit=true, AutoCommitTxn=OFF	never	never	never

Limitations and Workarounds

- The client setting of transaction isolation level is not propogated to the connectors. The transaction isolation level can be set on each XA connector, however this isolation level is fixed and cannot be changed at runtime for specific connections/commands.

Data Roles

Data roles, also called entitlements, are sets of permissions defined per VDB that dictate data access (create, read, update, delete). Data roles use a fine-grained permission system that Teiid will enforce at runtime and provide audit log entries for access violations - see Logging and Custom Logging for more.

Prior to applying data roles, you should consider restricting source system access through the fundamental design of your VDB. Foremost, Teiid 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 VDB.

If data role validation is enabled and data roles are defined in a VDB, then access permissions will be enforced by the Teiid 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 system functions that can be used for row-based and other authorization checks.

Tip	A VDB deployed without data roles is open for use by any authenticated user. If you want to ensure some attempt has been made at securing access, then set the data-roles-required configuration element to true via the CLI or in the standalone.xml on the teiid subsystem.

Permissions

Permissions or grants control access to data in several ways. There are simple access restrictions to READ, UPDATE, etc. down to a column level - note that the column or table metadata won't be visible to the user in JDBC/ODBC unless at user can read at least a single column.

You may also use permissions to filter and mask results, and constrain/check update values.

Table of Contents

- User Query Permissions
- Row and Column Based Security
- Row-Based Security
  - How Row-Based Conditions Are Applied
  - Considerations When Using Conditions
  - Limitations
- Column Masking
  - How Column Masks Are Applied
  - Considerations When Using Masking
  - Limitations

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 read to "model" will also grant read to "model.table", "model.table.column", etc. 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 only applied to the columns/tables/procedures in the user query - not to every resource 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.

Note	Unlike previous versions of Teiid, 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.

To process a *SELECT* statement or a stored procedure execution, the user account requires the following access rights:

- *READ*- on the Table(s) being accessed or the procedure being called.
- *READ*- on every column referenced.

To process an *INSERT* statement, the user account requires the following access rights:

- *CREATE*- on the Table being inserted into.
- *CREATE*- 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.
- *READ*- 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.
- *READ*- on every column referenced in the criteria.

To process a *EXEC/CALL* statement, the user account requires the following access rights:

- *EXECUTE (or READ)*- on the Procedure being executed.

To process any function, the user account requires the following access rights:

- *EXECUTE (or READ)*- 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 Teiid temporary table requires the following access rights:

- allow-create-temporary-tables attribute on any applicable role
- *CREATE,READ,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 at a more granular and consistent level the data returned to users.

See also XML Definition for examples of specifying data roles with row and column based security.

Row-Based Security

A permission against a fully qualified table/view/procedure may also specify a condition. Unlike the allow CRUD actions defined above, a condition is always applied - not just at the user query level. The condition can be any valid SQL 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.

How Row-Based Conditions Are Applied

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, i.e. "SELECT * FROM TBL WHERE blah **AND condition**". The condition will be present regardless of how the table/view is used in the query, whether via a union, join, etc.

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, etc. Inserts/updates against views are not checked with regards to the constraint. You may disable the insert/update constraint check by setting the condition constraint flag to false. This is typically only needed in circumstances when the condition cannot always be evaluated. However disabling the condition as a constraint simply drops the condition from consideration when logically evaluating the constraint. Any other condition constraints will still be evaluated.

Example non-constraint condition

```
<permission>
  <resource-name>modelName.tblName</resource-name>
  <condition constraint="false">column1=user()</condition>
</permission>
```

Across multiple applicable roles if more than one condition applies to the same resource, the conditions will be accumulated disjunctively via OR, i.e. "(condition1) **OR** (condition2) …". Therefore granting a permission with the condition "true" will allow users in that role to see all rows of the given resource.

Considerations When Using Conditions

Non-pushdown conditions may adversely impact performance, since their evaluation may inhibit pushdown of query constructs on top of the affected resource. Multiple conditions against the same resource should generally be avoided as any non-pushdown condition will cause the entire OR of conditions to not be pushed down. In some circumstances the insertion of permission conditions may require that the plan be altered with the addition of an inline view, which can result in adverse performance against sources that do not support inline views.

Pushdown of multi-row insert/update operations will be inhibited since the condition must be checked for each row.

In addition to managing permission conditions on a per-role basis, another approach is to add condition permissions would in an any authenticated role such that the conditions 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 row-based policy in effect such that all query plans can still be shared between users.

Handling of null values is up to the implementer of the data role and may require ISNULL checks to ensure that null values are allowed when a column is nullable.

Limitations

- 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 may 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.

How Column Masks Are Applied

Column masking is applied only against SELECTs. Column masking is applied logically after the affect of row based security. However since both views and source tables may have row and column based security, the actual view level masking may take place on top of source level masking. If the condition is specified along with the mask, then the effective mask expression effects 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".

Considerations When Using Masking

Non-pushdown masking conditions/expressions may adversely impact performance, since their evaluation may 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 adverse performance against sources that do not support 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.

Limitations

- In the event that two masks have the same order value, it is not well defined what order they are applied in.
- Masks or their 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	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.

Role Mapping

Each Teiid data role can be mapped to any number of container roles or any authenticated user. You may control role membership through whatever system the Teiid security domain login modules are associated with. The kit includes example files for use with the UsersRolesLoginModule - see teiid-security-roles.properties.

If you have an alternative security domain that a VDB should use, then set the VDB property security-domain to the relevant security domain.

It is possible for a user to have any number of container roles, which in turn imply a subset of Teiid data roles. Each applicable Teiid data role contributes cumulatively to the permissions of the user. No one role supersedes or negates the permissions of the other data roles.

XML Definition

Data roles are defined inside the `vdb.xml` file (inside the .vdb Zip archive under META-INF/vdb.xml). The "vdb.xml" file is checked against the schema file `vdb-deployer.xsd` , which can be found in the kit under docs/teiid/schema. This example will show a sample "vdb.xml" file with few simple data roles.

For example, if a VDB defines a table "TableA" in schema "modelName" with columns (column1, column2) - note that the column types do not matter. And we wish to define three roles "RoleA", "RoleB", and "admin" with following permissions:

1. RoleA has permissions to read, write access to TableA, but can not delete.
2. RoleB has permissions that only allow read access to TableA.column1
3. admin has all permissions

vdb.xml defining RoleA, RoleB, and Admin

```
<?xml version="1.0" encoding="UTF-8"?>
<vdb name="sample" version="1">

    <model name="modelName">
        <source name="source-name" translator-name="oracle" connection-jndi-name="java:myDS" />
    </model>

    <data-role name="RoleA">
        <description>Allow all, except Delete</description>

        <permission>
            <resource-name>modelName.TableA</resource-name>
            <resource-type>TABLE</resource-type>
            <allow-create>true</allow-create>
            <allow-read>true</allow-read>
            <allow-update>true</allow-update>
        </permission>

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

    </data-role>

    <data-role name="RoleB">
        <description>Allow read only</description>

        <permission>
            <resource-name>modelName.TableA</resource-name>
            <resource-type>TABLE</resource-type>
            <allow-read>true</allow-read>
        </permission>

        <permission>
            <resource-name>modelName.TableA.colum2</resource-name>
            <resource-type>COLUMN</resource-type>
            <allow-read>false</allow-read>
        </permission>

        <mapped-role-name>role2</mapped-role-name>
    </data-role>

    <data-role name="admin" grant-all="true">
        <description>Admin role</description>

        <mapped-role-name>admin-group</mapped-role-name>
    </data-role>
</vdb>
```

The above XML defined three data roles, "RoleA" which allows everything except delete on the table, "RoleB" that allows only read operation on the table, and the "admin" role with all permissions. Since Teiid uses deny by default, there is no explicit data-role entry needed for "RoleB". Note that explicit column permissions are not needed for RoleA, since the parent resource path, modelName.TableA, permissions still apply. RoleB however must explicitly disallow read to column2.

The "mapped-role-name" defines the container JAAS roles that are assigned the data role. For assigning roles to your users in the WildFly, check out the instructions for the selected Login Module. Check the "Admin Guide" for configuring Login Modules.

Using the grant-all option provides every permission on over object in the vdb. When importing a vdb and its roles, grant-all applies only to resources from the imported vdb.

Note	The optional resource-type element currently accepts LANGUAGE, SCHEMA, DATABASE, PROCEDURE, FUNCTION, TABLE, COLUMN. This property ensures that migration issues will be prevented when switching to DDL vdbs or dealing with multi-part table names.

Additional Role Attributes

You may also choose to allow any authenticated user to have a data role by setting the any-authenticated attribute value to true on data-role element.

The "allow-create-temporary-tables" data-role boolean attribute is used to explicitly enable or disable temporary table usage for the role. If it is left unspecified, then the value will be defaulted to false.

Temp Table Role for Any Authenticated

```
<data-role name="role" any-authenticated="true" allow-create-temporary-tables="true">
     <description>Temp Table Role for Any Authenticated</description>

     <permission>
         ...
     </permission>

</data-role>
```

Language Access

The following shows a vdb xml that allows the use of the javascript language. The allowed-languages property enables the languages use for any purpose in the vdb, while the allow-language permission allows the language to be used by users with RoleA.

vdb.xml allowing JavaScript access

```
<?xml version="1.0" encoding="UTF-8"?>
<vdb name="sample" version="1">

    <property name="allowed-languages" value="javascript"/>

    <model name="modelName">
        <source name="source-name" translator-name="oracle" connection-jndi-name="java:myDS" />
    </model>

    <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>

</vdb>
```

Row-Based Security

The following shows a vdb xml utilizing a condition to restrict access. The condition acts as both a filter and constraint. Even though RoleA opens up read/insert access to modelName.tblName, the base-role condition will ensure that only values of column1 matching the current user can be read or inserted. Note that here the constraint enforcement has been disabled.

vdb.xml allowing conditional access

```
<?xml version="1.0" encoding="UTF-8"?>
<vdb name="sample" version="1">

    <model name="modelName">
        <source name="source-name" translator-name="oracle" connection-jndi-name="java:myDS" />
    </model>

    <data-role name="base-role" any-authenticated="true">
        <description>Conditional access</description>

        <permission>
            <resource-name>modelName.tblName</resource-name>
            <condition constraint="false">column1=user()</condition>
        </permission>

    </data-role>

    <data-role name="RoleA">
        <description>Read/Insert access.</description>

        <permission>
            <resource-name>modelName.tblName</resource-name>
            <allow-read>true</allow-read>
            <allow-create>true</allow-create>
        </permission>

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

    </data-role>

</vdb>
```

Column Masking

The following shows a vdb xml utilizing column masking. Here the RoleA column1 mask takes precedence over the base-role mask, but only for a subset of the rows as specified by the condition. For users without RoleA, access to column1 will effectively be replaced with "CASE WHEN column1=user() THEN column1 END", while for users with RoleA, access to column1 will effectively be replaced with "CASE WHEN column2=’x’ THEN column1 WHEN TRUE THEN CASE WHEN column1=user() THEN column1 END END".

vdb.xml with column masking

```
<?xml version="1.0" encoding="UTF-8"?>
<vdb name="sample" version="1">

    <model name="modelName">
        <source name="source-name" translator-name="oracle" connection-jndi-name="java:myDS" />
    </model>

    <data-role name="base-role" any-authenticated="true">
        <description>Masking</description>
```

```
        <permission>
            <resource-name>modelName.tblName.column1</resource-name>
            <mask>CASE WHEN column1=user() THEN column1 END</mask>
        </permission>

    </data-role>

    <data-role name="RoleA">
        <description>Read/Insert access.</description>

        <permission>
            <resource-name>modelName.tblName</resource-name>
            <allow-read>true</allow-read>
            <allow-create>true</allow-create>
        </permission>

        <permission>
            <resource-name>modelName.tblName.column1</resource-name>
            <condition>column2='x'</condition>
            <mask order="1">column1</mask>
        </permission>

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

    </data-role>

</vdb>
```

Customizing

See the Developer’s Guide chapters on Custom Authorization Validators and [Teiid:Login Modules] for details on using an alternative authorization scheme.

System Schema

The built-in SYS and SYSADMIN schemas provide metadata tables and procedures against the current VDB.

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/accessible.

Unlike Teiid 8.x and prior releases when Data Roles are in use table/views and procedure metadata entries will not be visible if the user is not entitled to use the object. Tables/views/columns require the READ permission and procedures require the EXECUTE permission. All columns of a key must be accessible for the entry to be visible.

Note	If there is any caching of system metadata when data roles are in use, then visibility needs to be considered.

SYS Schema

System schema for public information and actions.

Table of Contents

- Tables/Views
  - SYS.Columns
  - SYS.DataTypes
  - SYS.KeyColumns
  - SYS.Keys
  - SYS.ProcedureParams
  - SYS.Procedures
  - SYS.FunctionParams
  - SYS.Functions
  - SYS.Properties
  - SYS.ReferenceKeyColumns
  - SYS.Schemas
  - SYS.Tables
  - SYS.VirtualDatabases
  - SYS.spatial_sys_ref
  - SYS.GEOMETRY_COLUMNS
- Procedures
  - SYS.getXMLSchemas
  - SYS.ArrayIterate

Tables/Views

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	Teiid runtime data type name
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
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	Teiid 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
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	Teiid 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
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)
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	Teiid 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
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	Teiid 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
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, etc) 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

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 will be set automatically by the relevant importer. If they are not set, they will be reported as 2 and 0 respectively. If client logic expects actual values, such as integration with GeoServer, then you may need to set these values manually.

Procedures

SYS.getXMLSchemas

DEPRECATED: Returns a resultset with a single column, schema, containing the schemas as xml.

```
SYS.getXMLSchemas(IN document string NOT NULL) RETURNS TABLE (schema xml)
```

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'.

SYSADMIN Schema

System schema for administrative information and actions.

Table of Contents

- Tables/Views
  - SYSADMIN.Usage
  - SYSADMIN.MatViews
  - SYSADMIN.VDBResources
  - SYSADMIN.Triggers
  - SYSADMIN.Views
  - SYSADMIN.StoredProcedures
- Procedures
  - SYSADMIN.isLoggable
  - SYSADMIN.logMsg
  - SYSADMIN.refreshMatView
  - SYSADMIN.refreshMatViewRow
  - SYSADMIN.refreshMatViewRows
  - SYSADMIN.setColumnStats
  - SYSADMIN.setProperty
  - SYSADMIN.setTableStats
  - SYSADMIN.matViewStatus
  - SYSADMIN.loadMatView
  - SYSADMIN.updateMatView

Tables/Views

SYSADMIN.Usage

This table supplies information about how views / 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 yet 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_ElementNam
e
      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 na
me')
    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

This 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.
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

This 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

This table provides the Triggers in the virtual database.

Column Name	Type	Description
VDBName	string	VDB name
SchemaName	string	Schema Name
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

This 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

This 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
```

Procedures

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. 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')
```

This will log the message 'some debug' at the default level DEBUG to the context org.something.

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.

See also SYSADMIN.loadMatView

```
SYSADMIN.refreshMatView(OUT RowsUpdated integer NOT NULL RESULT, IN ViewName string NOT NULL, IN Invalidate boo
lean NOT NULL DEFAULT 'false')
```

SYSADMIN.refreshMatViewRow

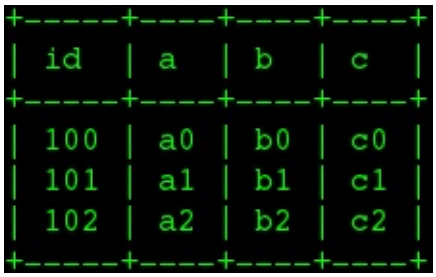
Refreshes a row in an internal materialized view.

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.

```
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 of 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')
```

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. See the Caching Guide for more.

```
SYSADMIN.refreshMatViewRows(OUT RowsUpdated integer NOT NULL RESULT, IN ViewName string NOT NULL, VARIADIC Key
object[] NOT NULL)
```

Example of 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'))
```

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.

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 "Va
lue" 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'
)
```

This 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.

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. A MetadataRepository must be configured to make a non-temporary metadata update persistent. See the Developer's Guide Runtime Metadata Updates section for more.

SYSADMIN.matViewStatus

matViewStatus is used to retrieve Materialized views' status 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 (TargetSchemaN
ame varchar(50), TargetName varchar(50), Valid boolean, LoadState varchar(25), Updated timestamp, Cardinality l
ong, 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.

```
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 may reference the view columns by qualified name, but all instances of '.' in the view name will be replaced by '_' as 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.

```
SYSADMIN.updateMatView(IN schemaName string NOT NULL, IN viewName string NOT NULL, IN refreshCriteria string) R
ETURNS 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''')
```

Translators

The Teiid Connector Architecture (TCA) provides Teiid with a robust mechanism for integrating with external systems. The TCA defines a common client interface between Teiid and an external system that includes metadata as to what SQL constructs are supported 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 Teiid and an external system, which is most commonly accessed through a JCA resource adapter. Refer to the Teiid Developers Guide for details on developing custom Translators and JCA resource adapters for use with Teiid.

Tip	The TCA is not the same as the JCA, the JavaEE Connector Architecture, although the TCA is designed for use with JCA resource adapters.

A Translator is typically paired with a particular JCA resource adapter. In instances where pooling, environment dependent configuration management, advanced security handling, etc. are not needed, then a JCA resource adapter is not needed. The configuration of JCA ConnectionFactories for needed resource adapters is not part of this guide, please see the Teiid Administrator Guide and the kit examples for configuring resource adapters for use in WildFly.

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, e.g. 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 1. **Base Execution Properties - shared by all translators**

Name	Description	Default
Immutable	Set to true to indicate that the source never changes.	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 is supported.	false
SupportsOuterJoins	Set to true to indicate that OUTER JOINs are supported.	false
SupportsFullOuterJoins	If outer joins are supported, true indicates that FULL OUTER JOINs are supported.	false
SupportsInnerJoins	Set to true to indicate that INNER JOINs are supported.	false
SupportedJoinCriteria	If joins are supported, defines what criteria may be used as the join criteria. May be one of (ANY, THETA, EQUI, or KEY).	ANY
MaxInCriteriaSize	If in criteria are supported, defines what the maximum number of in entries are per predicate. -1 indicates no limit.	-1

MaxDependentInPredicates	If in criteria are supported, 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 the direct query procedure is supported on the translator, this property indicates the name of the procedure.	native
SupportsDirectQueryProcedure	Set to true to indicate the translator supports the direct execution of commands	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 lobs (clob, blob, sql/xml) will be copied by the engine in a memory safe manner. Use this option if the source does not support memory safe lobs or you want to disconnect lobs from the source connection.	false
TransactionSupport	The highest level of transaction support. 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 supports metadata can be set through execution properties. If more control is needed, please consult the Developer's Guide.

There are no base importer settings.

Override Execution Properties

For all translators, you may override Execution Properties in the *vdb.xml* file.

Example Overriding of Translator Property

```
<model name="ora">
     <source name="ora" translator-name="oracle-override" connection-jndi-name="java:/oracle"/>
</model>

<translator name="oracle-override" type="oracle">
     <property name="RequiresCriteria" value="true"/>
</translator>
```

The above XML fragment is overriding the *oracle* translator and altering the behavior of *RequiresCriteria* property to true. Note that 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, i.e. $1 Note that 1 based indexing is used and that only IN parameters may be referenced. Dollar-sign integer is therefore reserved, but may be escaped with another $, i.e. $$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.". Example: importer.tableTypes

Name	Description	Default
autoCorrectColumnNames	Replace any usage of . in a column name with _ as the period character is not supported by Teiid in 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 Teiid name. For example use prod_%s to prefix all names with prod_.	

Amazon S3 Translator

The Amazon S3 translator, known by the type name *amazon-s3*, exposes stored procedures to leverage Amazon S3 object resources. The Web Service Data Source resource-adapter will be used for access. This will commonly be used with the TEXTTABLE or XMLTABLE table functions to use CSV or XML formatted data or read excel files, or other object files stored in the Amazon S3. This translator supports access to Amazon S3 using access-key and secret-key.

Usage

Here is sample VDB that is reading CSV file from Amazon S3 with name 'g2.txt' in the 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
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 SSE-C type encryption used, where customer supplies the encryption key, this key will be used for defining the "type" of encryption algorithm used. Supported are AES-256, AWS-KMS. 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	See override an execution property and the example below to set the properties.

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	Please note that bucket, region, accesskey, secretkey, encryption and encryptionkey are optional or nullable parameters in most of the methods provided. i.e. user do not need to provide them, if they are already configured using translator override properties as shown in above vdb example.

getTextFile(…)

Retrieves the given named object as text file from specified bucket and region 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 this URL will be 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 stream is true, then returned lobs may only be read once and will not typically be 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 this URL will be 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 stream is true, then returned lobs may only be read once and will not typically be 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', s
ecretkey=>'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 below procedure signature *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	currently *saveFile* does NOT support streaming/chuncked based upload of the contents. i.e. if you try to load very large objects there is risk of reaching out of memory issues. This method does not support SSE-C based security encryption either.

exmaples

```
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 s
ecretkey)
```

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 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>&amp;quot;fa44a7893b1735905bfcce59d9d9ae2e&amp;quot;</ETag>
        <Size>48</Size>
        <StorageClass>STANDARD</StorageClass>
    </Contents>
</ListBucketResult>
```

You can parse this into view using a example query like below

```
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;
```

When all properties like bucket, region, accesskey and secretkey are defined as translator override properties one can also issue simply

```
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 Teiid with enhancement request.

JCA Resource Adapter

The resource adapter for this translator provided through "Web Service Data Source", Refer to Admin Guide for configuration information.

Amazon SimpleDB Translator

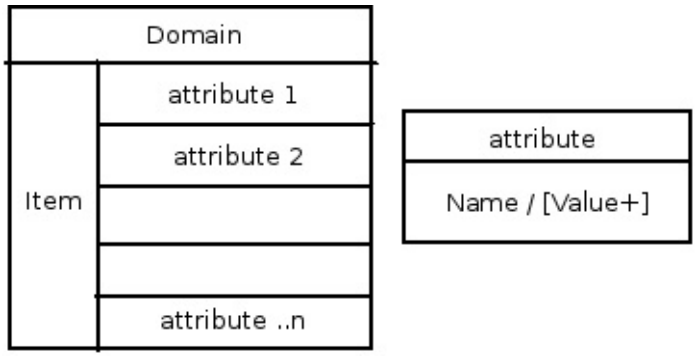
The Amazon SimpleDB Translator, known by the type name *simpledb*, exposes querying functionality to Amazon SimpleDB Data Sources.

Note	**"Amazon SimpleDB"** - Amazon SimpleDB is a web service for running queries on structured data in real time. This service works in close conjunction with Amazon Simple Storage Service (Amazon S3) and Amazon Elastic Compute Cloud (Amazon EC2), collectively providing the ability to store, process and query data sets in the cloud. These services are designed to make web-scale computing easier and more cost-effective for developers. Read more about it at http://aws.amazon.com/simpledb/

This translator provides an easy way connect to Amazon SimpleDB and provides relational way using SQL to add records from directly from user or from other sources that are integrated with Teiid. It also gives ability to read/update/delete existing records from SimpleDB store.

Usage

Amazon SimpleDB is hosted key/value store where a single key can contain host multiple attribute name/value pairs where where value can also be a multi-value. The data structure can be represented by



Based on above data structure, when you import the metadata from SimpleDB into Teiid, the constructs are aligned as below

Simple DB Name	SQL (Teiid)
Domain	Table
Item Name	Column (ItemName) Primary Key
attribute - single value	Column - String Datatype
attribute - multi value	Column - String Array Datatype

Since all attributes are by default are considered as string data types, columns are defined with string data type.

Note	If you did modify data type be other than string based, be cautioned and do not use those columns in comparison queries, as SimpleDB does only lexicographical matching. To avoid it, set the "SearchType" on that column to "UnSearchable".

An Example VDB that shows SimpleDB translator can be defined as

```
<vdb name="myvdb" version="1">
    <model name="simpledb">
```

```
        <source name="node" translator-name="simpledb" connection-jndi-name="java:/simpledbDS"/>
    </model>
<vdb>
```

The translator does NOT provide a connection to the SimpleDB. For that purpose, Teiid has a JCA adapter that provides a connection to SimpleDB using Amazon SDK Java libraries. To define such connector, see Amazon SimpleDB Data Sources or see an example in "<jboss-as>/docs/teiid/datasources/simpledb"

Properties

The Amazon SimpleDB Translator currently has no import or execution properties.

Capabilities

The Amazon SimpleDB Translator supports SELECT statements with a restrictive set of capabilities including: comparison predicates, IN predicates, LIMIT and ORDER BY. Insert, update, delete are also supported.

Queries on Attributes with Multiple Values

Attributes with multiple values will defined as string array type. So this column is treated SQL Array type. The below table shows SimpleDB way of querying to Teiid way to query. The queries are based on http://docs.aws.amazon.com/AmazonSimpleDB/latest/DeveloperGuide/RangeValueQueriesSelect.html

SimpleDB Query	Teiid Query
select * from mydomain where Rating = '4 stars' or Rating = '**'	select * from mydomain where Rating = ('4 stars','**')
select * from mydomain where Keyword = 'Book' and Keyword = 'Hardcover'	select * from mydomain where intersection(Keyword,'Book','Hardcover')
select * from mydomain where every(Rating) = '**'	select * from mydomain where every(Rating) = '**'

With Insert/Update/Delete you write prepare statements or you can write SQL like

```
INSERT INTO mydomain (ItemName, title, author, year, pages, keyword, rating) values ('0385333498', 'The Sirens of Titan', 'Kurt Vonnegut', ('1959'), ('Book', Paperback'), ('*****','5 stars','Excellent'))
```

Direct Query Support

Note	This feature is turned off by default because of the security risk this exposes to execute any command against the source. To enable this feature, override the execution property called SupportsDirectQueryProcedure to true.

Tip	By default the name of the procedure that executes the queries directly is called native. Override the execution property DirectQueryProcedureName to change it to another name.

The SimpleDB translator provides a procedure to execute any ad-hoc simpledb query directly against the source without Teiid parsing or resolving. Since the metadata of this procedure's results are not known to Teiid, they are returned as an object array. ARRAYTABLE can be used construct tabular output for consumption by client applications. Direct query supported for "select" based calls.

```
SELECT X.*
```

```
   FROM simpledb_source.native('SELECT firstname, lastname FROM users') n, ARRAYTABLE(n.tuple COLUMNS firstname
string, lastname string) AS X
```

JCA Resource Adapter

The Teiid specific Amazon SimpleDB Resource Adapter should be used with this translator. See Amazon SimpleDB Data Sources for connecting to SimpleDB.

Apache Accumulo Translator

The Apache Accumulo Translator, known by the type name *accumulo*, exposes querying functionality to Accumulo Data Sources. Apache Accumulo is a sorted, distributed key value store with robust, scalable, high performance data storage and retrieval system. This translator provides an easy way connect to Accumulo system and provides relational way using SQL to add records from directly from user or from other sources that are integrated with Teiid. It also gives ability to read/update/delete existing records from Accumulo store. Teiid has capability to pass-in logged in user's roles as visibility properties to restrict the data access.

Tip	**"versions"** - The development was done using Accumulo 1.5.0, Hadoop 2.2.0 and Zookeeper 3.4.5

Note	This document assumes that user is familiar with Accumulo source and has basic understanding of how Teiid works. This document only contains details about Accumulo translator.

Intended Usecases

The usage Accumulo translator can be highly dependent on user's usecase(s). Here are some common scenarios.

- Accumulo source can be used in Teiid, to continually add/update the documents in the Accumulo system from other sources automatically.
- Access Accumulo through SQL interface.
- Make use of cell level security through enterprise roles.
- Accumulo translator can be used as an indexing system to gather data from other enterprise sources such as RDBMS, Web Service, SalesForce etc, all in single client call transparently with out any coding.

Usage

Apache Accumulo is distributed key value store with unique data model. It allows to group its key-value pairs in a collection called "table". The key structure is defined as

<table>
<tr><th colspan="4">Key</th><th rowspan="2">Value</th></tr>
<tr><th rowspan="2">Row ID</th><th colspan="3">Column</th><th rowspan="2">Timestamp</th></tr>
<tr><th>Family</th><th>Qualifier</th><th>Visibility</th></tr>
</table>

Based on above information, one can define a schema representing Accumulo table structures in Teiid using DDL with help of metadata extension properties defined below. Since no data type information is defined on the columns, by default all columns are considered as string data types. However, during modeling of the schema, one can use various other data types supported through Teiid to define a data type of column, that user wishes to expose as.

Once this schema is defined and exposed through VDB in a Teiid database, and Accumulo Data Sources is created, the user can issue "INSERT/UPDATE/DELETE" based SQL calls to insert/update/delete records into the Accumulo, and issue "SELECT" based calls to retrieve records from Accumulo. You can use full range of SQL with Teiid system integrating other sources along with Accumulo source.

By default, Accumulo table structure is flat can not define relationships among tables. So, a SQL JOIN is performed in Teiid layer rather than pushed to source even if both tables on either side of the JOIN reside in the Accumulo. Currently any criteria based on EQUALITY and/or COMPARISON using complex AND/OR clauses are handled by Accumulo translator and will be properly executed at source.

An Example VDB that shows Accumulo translator can be defined as

```
<vdb name="myvdb" version="1">
    <model name="accumulo">
        <source name="node-one" translator-name="accumulo" connection-jndi-name="java:/accumuloDS"/>
    </model>
<vdb>
```

The translator does NOT provide a connection to the Accumulo. For that purpose, Teiid has a JCA adapter that provides a connection to Accumulo using Accumulo Java libraries. To define such connector, see Accumulo Data Sources or see an example in "<jboss-as>/docs/teiid/datasources/accumulo"

Properties

Accumulo translator is capable of traversing through Accumulo table structures and build a metadata structure for Teiid translator. The schema importer can understand simple tables by traversing a single ROWID of data, then looks for all the unique keys, based on it it comes up with a tabular structure for Accumulo based table. Using the following import properties, you can further refine the import behavior.

Import Properties

Property Name	Description	Required	Default
ColumnNamePattern	How the column name should be formed	false	{CF}_{CQ}
ValueIn	Where the value for column is defined CQ or VALUE	false	{VALUE}

Note	{CQ}, {CF}, {ROWID} are expressions that you can use to define above properties in any pattern, and respective values of Column Qualifer, Column Familiy or ROWID will be replaced at import time. ROW ID of the Accumulo table, is automatically created as ROWID column, and will be defined as Primary Key on the table.

You can also define the metadata for the Accumulo based model using DDL. When doing such exercise, the Accumulo Translator currently defines following extended metadata properties to be defined on its Teiid schema model to guide the translator to make proper decisions. The following properties are described under NAMESPACE "http://www.teiid.org/translator/accumulo/2013", for user convenience this namespace has alias name *teiid_accumulo* defind in Teiid. To define a extension property use expression like "teiid_accumulo:{property-name} value". All the properties below are intended to be used as OPTION properties on COLUMNS. See DDL Metadata for more information on defining DDL based metadata.

Extension Metadata Properties

Property Name	Description	Required	Default
CF	Column Family	true	none
CQ	Column Qualifier	false	empty

VALUE-IN	Value of column defined in. Possible values (VALUE, CQ)	false	VALUE

How to use above Properties

Say for example you have a table called "User" in your Accumulo instance, and doing a scan returned following data

```
root@teiid> table User
root@teiid User> scan
  1 name:age []    43
  1 name:firstname []   John
  1 name:lastname []    Does
  2 name:age []    10
  2 name:firstname []   Jane
  2 name:lastname []    Smith
  3 name:age []    13
  3 name:firstname []   Mike
  3 name:lastname []    Davis
```

If you used the default importer from the Accumulo translator (like the VDB defined above), the table generated will be like below

```
CREATE FOREIGN TABLE "User" (
    rowid string OPTIONS (UPDATABLE FALSE, SEARCHABLE 'All_Except_Like'),
    name_age string OPTIONS (SEARCHABLE 'All_Except_Like', "teiid_accumulo:CF" 'name', "teiid_accumulo:CQ" 'age'
, "teiid_accumulo:VALUE-IN" '{VALUE}'),
    name_firstname string OPTIONS (SEARCHABLE 'All_Except_Like', "teiid_accumulo:CF" 'name', "teiid_accumulo:CQ"
 'firstname', "teiid_accumulo:VALUE-IN" '{VALUE}'),
    name_lastname string OPTIONS (SEARCHABLE 'All_Except_Like', "teiid_accumulo:CF" 'name', "teiid_accumulo:CQ"
'lastname', "teiid_accumulo:VALUE-IN" '{VALUE}'),
    CONSTRAINT PK0 PRIMARY KEY(rowid)
) OPTIONS (UPDATABLE TRUE);
```

You can use "Import Property" as "ColumnNamePattern" as "{CQ}" will generate the following (note the names of the column)

```
CREATE FOREIGN TABLE "User" (
    rowid string OPTIONS (UPDATABLE FALSE, SEARCHABLE 'All_Except_Like'),
    age string OPTIONS (SEARCHABLE 'All_Except_Like', "teiid_accumulo:CF" 'name', "teiid_accumulo:CQ" 'age', "t
eiid_accumulo:VALUE-IN" '{VALUE}'),
    firstname string OPTIONS (SEARCHABLE 'All_Except_Like', "teiid_accumulo:CF" 'name', "teiid_accumulo:CQ" 'fi
rstname', "teiid_accumulo:VALUE-IN" '{VALUE}'),
    lastname string OPTIONS (SEARCHABLE 'All_Except_Like', "teiid_accumulo:CF" 'name', "teiid_accumulo:CQ" 'las
tname', "teiid_accumulo:VALUE-IN" '{VALUE}'),
    CONSTRAINT PK0 PRIMARY KEY(rowid)
) OPTIONS (UPDATABLE TRUE);
```

respectively if the column name is defined by Column Family, you can use "ColumnNamePattern" as "{CF}", and if the value for that column exists in the Column Qualifier then you can use "ValueIn" as "{CQ}". Using import properties you can dictate how the table should be modeled.

JCA Resource Adapter

The Teiid specific Accumulo Resource Adapter should be used with this translator. See Accumulo Data Sources for connecting to a Accumulo Source.

Native Queries

Currently this feature is not applicable. Based on user demand Teiid could expose a way for user to submit a MAP-REDUCE job.

Direct Query Procedure

This feature is not applicable for this translator.

Apache SOLR Translator

The Apache SOLR Translator, known by the type name *solr*, exposes querying functionality to Solr Data Sources. Apache Solr is a search engine built on top of Apache Lucene for indexing and searching. This translator provides an easy way connect to existing or a new Solr search system, and provides way to add documents/records from directly from user or from other sources that are integrated with Teiid. It also gives ability to read/update/delete existing documents from Solr Search system.

Properties

The Solr Translator currently has no import or execution properties. It does not define any extension metadata.

Intended Usecases

The usage Solr translator can be highly dependent on user's usecase(s). Here are some common scenarios.

- Solr source can be used in Teiid, to continually add/update the documents in the search system from other sources automatically.
- If the search fields are stored in Solr system, this can be used as very low latency data retrieval for serving high traffic applications.
- Solr translator can be used as a fast full text search. The Solr document can contain only the index information, then the results as an inverted index to gather target full documents from the other enterprise sources such as RDBMS, Web Service, SalesForce etc, all in single client call transparently with out any coding.

Usage

Solr search system provides searches based on indexed search fields. Each Solr instance is typically configured with a single core that defines multiple fields with different type information. Teiid metadata querying mechanism is equipped with "Luke" based queries, that at deploy time of the VDB use this mechanism to retrieve all the stored/indexed fields. Currently Teiid does NOT support dynamic fields and non-stored fields. Based on retrieved fields, Solr translator exposes a single table that contains all the fields. If a field is multi-value based, it's type is represented as Array type.

Once this table is exposed through VDB in a Teiid database, and Solr Data Sources is created, the user can issue "INSERT/UPDATE/DELETE" based SQL calls to insert/update/delete documents into the Solr, and issue "SELECT" based calls to retrieve documents from Solr. You can use full range of SQL with Teiid system integrating other sources along with Solr source.

The Solr Translator supports SELECT statements with a restrictive set of capabilities including: comparison predicates, IN predicates, LIMIT and Order By.

An Example VDB that shows Solr translator can be defined as

```
<vdb name="search" version="1">
    <model name="solr">
        <source name="node-one" translator-name="solr" connection-jndi-name="java:/solrDS"/>
    </model>
<vdb>
```

The translator does NOT provide a connection to the Solr. For that purpose, Teiid has a JCA adapter that provides a connection to Solr using the SolrJ Java library. To define such connector, see Solr Data Sources or see an example in "<jboss-as>/docs/teiid/datasources/solr"

JCA Resource Adapter

The Teiid specific Solr Resource Adapter should be used with this translator. See Solr Data Sources for connecting to a Solr Search Engine.

Native Queries

This feature is not applicable for Solr translator.

Direct Query Procedure

This feature is not available for Solr translator currently.

Cassandra Translator

The Cassandra Translator, known by the type name *cassandra*, exposes querying functionality to Cassandra Data Sources. The translator translates Teiid push down commands into Cassandra CQL.

Properties

The Cassandra Translator currently has no import or execution properties.

Usage

The Cassandra Translator supports INSERT/UPDATE/DELETE/SELECT statements with a restrictive set of capabilities including: count(*), comparison predicates, IN predicates, and LIMIT. Only indexed columns are searchable. Consider a custom extension or create an enhancement request should your usage require additional capabilities.

Cassandra updates always return an update count of 1 per update regardless of the number of rows affected.

Cassandra inserts are functionally upserts, that is if a given row exists it will be updated rather than causing an exception.

JCA Resource Adapter

The Teiid specific Cassandra Resource Adapter should be used with this translator. See Cassandra Data Sources for connecting to a Cassandra cluster.

Native Queries

Cassandra source procedures may be created using the teiid_rel:native-query extension - see Parameterizable Native Queries. 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 this feature, override the execution property called _SupportsDirectQueryProcedure to true.

By default the name of the procedure that executes the queries directly is called **native**. Override the execution property _DirectQueryProcedureName to change it to another name.

The Cassandra translator provides a procedure to execute any ad-hoc CQL query directly against the source without Teiid parsing or resolving. Since the metadata of this procedure's results are not known to Teiid, they are returned as an object array. ARRAYTABLE can be used construct tabular output for consumption by client applications.

Example CQL Direct Query

```
SELECT X.*
  FROM cassandra_source.native('SELECT firstname, lastname FROM users WHERE birth_year = $1 AND country = $2 AL
LOW FILTERING', 1981, 'US') n,
       ARRAYTABLE(n.tuple COLUMNS firstname string, lastname string) AS X
```

Couchbase Translator

The Couchbase Translator, known by the type name *couchbase*, exposes querying functionality to Couchbase Data Sources. The Couchbase Translator provide a SQL Integration solution for integrating Couchbase JSON document with relational model, which allows applications to use normal SQL queries against Couchbase Server, translating standard SQL-92 queries into equivalent N1QL client API calls. The translator translates Teiid push down commands into Couchbase N1QL.

Table of Contents

- Usage
- JCA Resource Adapter
- Execution Properties
- Schema Definition
  - Generating a Schema
  - Creating a Schema
  - An example of Schema Generation
- Procedures
  - Native Queries
  - getDocuments
  - getDocument

Usage

The Couchbase Translator supports INSERT, UPSERT, UPDATE, DELETE, SELECT and bulk INSERT statements with a restrictive set of capabilities including: count(*), comparison predicates, Order By, Group By, LIMIT etc. Consider a custom extension or create an enhancement request should your usage require additional capabilities.

JCA Resource Adapter

The Teiid specific Couchbase Resource Adapter should be used with this translator. See Couchbase Data Sources for connecting to a Couchbase cluster.

Execution Properties

Use the translator override mechanism to supply the following properties.

Name	Description	Default
UseDouble	Use double rather than allowing for more precise types, such as long, bigdecimal, and biginteger. This affects both import and execution. See the issue that describes problems with Couchbase and precision loss.	false

Schema Definition

Couchbase is able to store data that does not follow the rules of data typing and structure that apply to traditional relational tables and columns. Couchbase data is organized into buckets(keyspaces) and documents.

Logical Hierarchy of Couchbase Cluster

The document in a keyspace are structureless, it may have complex structure, like contain nested object, nested arrays, or arrays of differently-typed elements.

Note	The datastores are higher level abstraction, but the Couchbase Translator focus on one specific namespace, all documents in a namespace across different keyspaces will be map to tables of Teiid source metadata.

Because Teiid metadata/traditional JDBC toolsets might not support these data structures, the data needs to be mapped to a relational form. To achieve this, the Couchbase Translator provide a way to automatically generates schema during VDB deploying. Refer to Generating a Schema for more details.

Alternatively, create the schema manually in a Teiid Source module are supported, creating a schema should base on the sample rules of generating a schema. Refer to Creating a Schema for more details.

Note	Use Generating a Schema are recommend.

Generating a Schema

Schema Generation is a way that the Couchbase Translator sample some data from a Couchbase cluster(namespace), and scan these documents data, generate a data typing and structure based schema that is needed for Teiid or traditional JDBC toolsets. The Importer Properties are used to control the behavior of data sampling.

The generated schema are tables and procedures, the procedures provide additional flexibility to execute native query; the tables are used to map to documents in a specific namespace. There are two kinds of table,

- Regular Table - map to a keyspace in a couchbase(namespace)
- Array Table - map to a array in any documents

A table option used to differentiate Regular Table and Array Table, refer to Additional Table Options for details.

The principle use to generate schema are following:

- Basically, a keyspace be map to a table, keyspace name is the table name, all documents' no-array attribute are column names, each document are a row in table. if `TypeNameList` defined, a keyspace may map to several tables, all same type referenced values are table names, all same type value referenced no-array attribute are map to column names correspondently. If multiple keyspaces has same typed value, the typed value table name will add each keyspace as prefix. For example,

```
TypeNameList=`default`:`type`,`default2`:`type`
```

both default and default2 has document defined {"type": "Customer"}, then the default's table name is 'Customer', default2's table name is 'default2_Customer'.

- Each generate table has a documentID column map to a couchbase document ID, the documentID in Regular Table play a role as primary key, the documentID in Array Table play a role as foreign key.
- Any of array in documents will be map to a Array Table, array index, array item or nested object item attribute are column names. If array contains differently-typed elements and no elements are object, all elements be map to same column with Object type; If array contains object, all object attribute be map to column names, and reference value data type be map to column data type;

- Each Array Table has at least one index column with the suffix `_idx` to indicate the position of the element within the array. If the dimension of array large than 1, multiple index column are created, the column name with explicity dimension identity `_dimX` , separated by underscore character. For example, a three dimension nested array document

```
"default": {"nested": [[["dimension 3"]]]}
```

the index columns might like: default_nested_idx, default_nested_dim2_idx, default_nested_dim2_dim3_idx.

- Each Table must define a NAMEINSOURCE to indicate the keyspace name or he path pattern in couchbase, the NAMEINSOURCE of Regular Table are keyspacename, the NAMEINSOURCE of Array Table are path pattern with square brackets suffix to indicate dimension of nested array. Use above three dimension nested array document as example, the NAMEINSOURCE of table might be `default` .nestedArray[][][].
- Each no documentID, no array index columns must be define a NAMEINSOURCE to indicate the path pattern in couchbase, the dot are use to separate the paths. For example, the `p_asia` are nested object attribute of a document in keyspace `travel-sample` :

```
default:`travel-sample`/geo/`p_asia`
```

the `p_asia` referenced column must define a NAMEINSOURCE with value `travel-sample` .geo. `p_asia` .

The Array Table column's NAMEINSOURCE must use a square brackets for each hierarchy level in which dimension the array is nested. For example, the `nestedArray` are nested array attribute of a document in keyspace `travel-sample` , it's dimension 3 nested array at least has two items, dimension 4 nested array at least has two items:

```
default:`travel-sample`/nestedArray[0][0][1][1]
```

the dimension 4 nested array coulmn must define a NAMEINSOURCE with value `travel-sample` .nestedArray[][][][]. If dimension 4 item has object item, then the coulmn NAMEINSOURCE might be `travel-sample` .nestedArray[][][][].id, `travel-sample` .nestedArray[][][][]. `address_name` , etc.

- If a table name defined by TypeNameList, another NAMEDTYPEPAIR option are used to define the type attribute, more details refer to Additional Table Options.

Importer Properties

To ensure consistent support for your Couchbase data, use the importer properties to do futher defining in shcema generation.

An example of importer properties

```
<model name="CouchbaseModel">
    <property name="importer.sampleSize" value="100"/>
    <property name="importer.typeNameList" value="`test`:`type`"/>
    <source name="couchbase" translator-name="translator-couchbase" connection-jndi-name="java:/couchbaseDS"/>
</model>
```

Name	Description
sampleSize	Set the SampleSize property to the number of documents per buckets that you want the connector to sample the documents data.
sampleKeyspaces	A comma-separate list of the keyspace names, used to fine-grained control which keyspaces should be mapped, by default map all keyspaces. The smaller scope of keyspaces, the larger sampleSize, if user focus on specific keyspace, and want more precise metadata, this property is recommended.

typeNameList	A comma-separate list of key/value pair that the buckets(keyspaces) use to specify document types. Eacl list item must be a bucket(keyspace) name surrounded by back quotes, a colon, and an attribute name surrounded by back quotes. .Syntax of typeNameList
	`` `KEYSPACE`:`ATTRIBUTE`,`KEYSPACE`:`ATTRIBUTE`,`KEYSPACE`:`ATTRIBUTE` ``
	• KEYSPACE - the keyspaces must be under same namespace it either can be different one, or are same one.
	• ATTRIBUTE - the attribute must be non object/array, resident on the root of keyspace, and it's type should be equivalent String. If a typeNameList set a specifc bucket(keyspace) has multiple types, a a document has all these types, the first one will be chose.
	For example, the TypeNameList below indicates that the buckets(keyspaces) test, default, and beer-samp use the type attribute to specify the type of each document, during schema generation, all type reference value will be treated as table name.
	`` TypeNameList=`test`:`type`,`default`:`type`,`beer-sample`:`type` ``
	The TypeNameList below indicates that the bucket(keyspace) test use type, name and category attribute specify the type of each document, during schema generation, the teiid connector scan the documents under test, if a document has attribute as any of type, name and category, it's referenced value will be treated as table name.
	`` TypeNameList=`test`:`type`,`test`:`name`,`test`:`category` ``

Additional Table Options

Name	Description
teiid_couchbase:NAMEDTYPEPAIR	A `NAMEDTYPEPAIR` OPTION in table declare the name of typed key/value pair. This option is used once the typeNameList importer property is used and the table is typeName referenced table.
teiid_couchbase:ISARRAYTABLE	A `ISARRAYTABLE` OPTION in table used to differentiate the array table and regular table. • A regular table represent data from collections of Couchbase documents. Documents appear as rows, and all attributes that are not arrays appear as columns. In each table, the primary key column named as documentID that that identifies which Couchbase document each row comes from. If no typed name defined the table name is the keyspace name, but in the Couchbase layer, the name of the table will be translate to keyspace name. • If a table defined the `ISARRAYTABLE` OPTION, then it provide support for arrays, each array table contains the data from one array, and each row in the table represents an element from the array. If an element contains an nested array, an additional virtual tables as needed to expand the nested data. In each array table there also has a documentID column play as a foreign key that identifies the Couchbase document the array comes from and references the documentID from normal table. An index column (with the suffix _IDX in its name) to indicate the position of the element within the array.

Creating a Schema

Creating a schema should strict base on the principles listed in Generating a Schema.

Couchbase supported Teiid types are String, Boolean, Integer, Long, Double, BigInteger, and BigDecimal. Creating a source model with other types is not fully supported.

Each table is expected to have a document ID column. It may be arbitrarily named, but it needs to be a string column marked as the primary key.

An example of Schema Generation

The following example shows the tables that the Couchbase connector would generate if it connected to a Couchbase, the keyspace named `test` under namespace `default` contains two kinds of documents named `Customer` and `Order`.

The `Customer` document is of type Customer and contains the following attributes. The SavedAddresses attribute is an array.

```
{
  "ID": "Customer_12345",
  "Name": "John Doe",
  "SavedAddresses": [
    "123 Main St.",
    "456 1st Ave"
  ],
  "type": "Customer"
}
```

The `Order` document is of type Order and contains the following attributes. The CreditCard attribute is an object, and the Items attribute is an array of objects.

```
{
  "CreditCard": {
    "CVN": 123,
    "CardNumber": "4111 1111 1111 111",
    "Expiry": "12/12",
    "Type": "Visa"
  },
  "CustomerID": "Customer_12345",
  "Items": [
    {
      "ItemID": 89123,
      "Quantity": 1
    },
    {
      "ItemID": 92312,
      "Quantity": 5
    }
  ],
  "Name": "Air Ticket",
  "type": "Order"
}
```

When the VDP deploy and load metedata, the connector exposes these collections as two tables show as below:

Customer

documentID	ID	type	Name
customer-1	Customer_12345	Customer	Kylin Soong

Order

documentID	CustomerID	type	CreditCard_CardNumber	CreditCard_Type	CreditCard_CVN	CreditCard_Expiry	Name
order-1	Customer_12345	Order	4111 1111 1111 111	Visa	123	12/12	Air Ticket

The SavedAddresses array from the Customer and the Items array from the Order document do not appear in above table. Instead, the following tables are generated for each array:

Customer_SavedAddresses

documentID	Customer_SavedAddresses_idx	Customer_SavedAddresses
customer-1	0	123 Main St.
customer-1	1	456 1st Ave

Order_Items

documentID	Oder_Items_idx	Oder_Items_Quantity	Oder_Items_ItemID
order-1	0	1	89123
order-1	1	5	92312

Procedures

Native Queries

Couchbase source procedures may be created using the teiid_rel:native-query extension - see Parameterizable Native Queries. 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.

Example of executing N1QL directly

```
EXEC CouchbaseVDB.native('DELETE FROM test USE KEYS ["customer-3", "order-3"]')
```

getDocuments

Returns the json documents that match the given document id or id pattern as BLOBs.

```
getDocuments(id, keyspace)
```

- id - The document id or SQL like pattern of what documents to return, for example, the '%' sign is used to define wildcards (missing letters) both before and after the pattern.
- keyspace - The keyspace name used to retrieve the documents.

Example of getDocuments()

```
call getDocuments('customer%', 'test')
```

getDocument

Returns a json document that match the given document id as BLOB.

```
getDocument(id, keyspace)
```

- id - The document id of what document to return.
- keyspace - The keyspace name used to retrieve the document.

Example of getDocument()

```
call getDocument('customer-1', 'test')
```

Delegating Translators

Translator "delegator"

A translator by name "delegator" is available in core Teiid installation, that can be used to modify the capabilities of a existing translator. Often times for debugging purposes or in special situations, one may require to either turn on/off certain capability of translator. For example, assume Hive database in their latest version supporting the ORDER BY construct, however Teiid's current version of the Hive translator does not have this capability, you can use the "delegator" translator to turn ON the "ORDER BY" support without actually writing any code. Sometimes you may want to do the reverse, you want turn off certain capability to produce a better plan. In these situations, you can use this translator.

To use this translator, you need to define this translator in the VDB, as a shown in the below VDB. The below example overriding the "hive" translator and turning off the ORDER BY support.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<vdb name="myvdb" version="1">

    <model name="mymodel">
        <source name="source" translator-name="hive-delegator" connection-jndi-name="java:hive-ds"/>
    </model>

    <!-- the below it is called translator overriding, where you can set different properties -->
    <translator name="hive-delegator" type="delegator" />
        <property name="delegateName" value="hive" />
        <property name="supportsOrderBy" value="false"/>
   </translator>
</vdb>
```

You can override any/all the translator capabilities defined here Translator Capabilities as Execution Properties to override. Example of "supportsOrderBy" is shown in above example.

Extending the "delegator" translator

You may create a delegating translator by extending the `org.teiid.translator.BaseDelegatingExecutionFactory` . Once your classes are then packaged as a custom translator, you will be able to 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. The difference here from previous "delegator" translator is, you can hard code the capabilities instead of defining as configuration inside the -vdb.xml, as well as override methods to provide alternate behavior.

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

Lets say you are currently using "oracle" translator in your VDB, you want to intercept the calls going through this translator, then you first write a custom delegating translator like

```
@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..
    }
}
```

Now deploy this translator in Teiid engine. Then in your -vdb.xml or .vdb file define like below.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<vdb name="myvdb" version="1">

    <model name="mymodel">
        <source name="source" translator-name="oracle-interceptor" connection-jndi-name="java:oracle-ds"/>
    </model>

    <!-- the below it is called translator overriding, where you can set different properties -->
    <translator name="orcle-interceptor" type="interceptor" />
        <property name="delegateName" value="oracle" />
   </translator>
</vdb>
```

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 your VDB. Now any calls going into this VDB model’s translator will be intercepted by YOUR code to do whatever you want to do.

File Translator

The file translator, known by the type name *file*, exposes stored procedures to leverage file system resources exposed by the File Data Source and the FTP Data Source. It will commonly be used with the TEXTTABLE or XMLTABLE table functions to use CSV or XML formatted data.

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 Teiid, see also TO_CHARS and TO_BYTES.	The system default encoding
ExceptionIfFileNotFound	Throw an exception in getFiles or getTextFiles if the specified file/directory does not exist.	true (false prior to 8.2)

Tip	See override an execution property and the example below to set the properties.

VDB XML Override Example

```
<model name="file">
    <source name="file" translator-name="file-override" connection-jndi-name="java:/file"/>
</model>

<translator name="file-override" type="file">
    <property name="Encoding" value="ISO-8859-1"/>
    <property name="ExceptionIfFileNotFound" value="false"/>
</translator>
```

Usage

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 will be returned. If the path matches a single file, it will be returned.

The '*' character will be treated as a wildcard to match any number of characters in the path name - 0 or matching files will be returned.

If '*' is not used and the path doesn’t exist and ExceptionIfFileNotFound is true, then an exception will be 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 CLOB(s) matching the given path and pattern.

```
call getTextFiles('path/*.ext')
```

All the same files a getFiles will be retrieved, the only difference is that the results will be CLOB values using 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. Or if the file cannot be deleted an exception will be thrown.

NOTE **Native queries** - Native or direct query execution is not supported on the File Translator.

JCA Resource Adapter

The resource adapter for this translator provided through "File Data Source", Refer to Admin Guide for configuration information.

Google Spreadsheet Translator

The *google-spreadsheet* translator is used to connect to a Google Spreadsheet. To use the Google Spreadsheet Translator you need to configure and deploy the Google JCA connector - see the Admin Guide.

The query approach expects the data in the worksheet to be in a specific format. Namely:

- Any column that has data is queryable.
- 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 interchanably. Where possible the translator may provide a warning or throw an exception if there may be a confusion of null vs. empty strings.
- If the first row is present and contains string values, then it will be 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) are loaded upon translator start up. If you make any changes in data types, it is advisable to restart your vdb.

The translator supports queries against a single sheet. It supports ordering, aggregation, basic predicates, and most of the functions supported by the spreadsheet query language.

There are no google-spreadsheet importer settings, but it can provide metadata for VDBs.

Warning	A sheet with a header that is defined in Teiid, which later has all data rows removed is no longer valid for access through Teiid. The google api will treat the header as a data row at that point and thus queries will no longer be valid.

Warning	Non-string fields are updated using the canonical Teiid SQL value - in cases where the spreadsheet is using a non-conforming locale, consider disallowing updates. See also TEIID-4854 and the allTypesUpdatable import property below.

Importer Properties

- *allTypesUpdatable*- Set to true to mark all columns as updatable. Set to false to enable update only on string/boolean columns, which are not affected by TEIID-4854. Defaults to true.

JCA Resource Adapter

The Teiid specific Google Spreadsheet Data Sources Resource Adapter should be used with this translator.

Native Queries

Google spreadsheet source procedures may be created using the teiid_rel:native-query extension - see Parameterizable Native Queries. 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. See the Select format below.

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 this feature, override the execution property called _SupportsDirectQueryProcedure to true.

Tip	By default the name of the procedure that executes the queries directly is called **native**. Override the execution property _DirectQueryProcedureName to change it to another name.

The Google spreadsheet translator provides a procedure to execute any ad-hoc query directly against the source without any Teiid parsing or resolving. Since the metadata of this procedure's execution results are not known to Teiid, they are returned as an object array. ARRAYTABLE can be used construct tabular output for consumption by client applications. Teiid exposes this procedure with a simple query structure as below:

Select

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 semi-colon(;) separated name value pairs of following properties to execute the procedure:

Property	Description	Required
worksheet	Google spreadsheet name	yes
query	spreadsheet query	yes
limit	number rows to fetch	no
offset	offset of rows to fetch from limit or beginning	no

Infinispan Translator

The Infinispan translator, known by the type name *"infinispan-hotrod"* exposes the Infinispan cache store to be queried using SQL language, and it uses HotRod protocol to connect the remote Infinispan cluster farm. This translator does NOT work with any arbitary key/value mappings in the Infinispan. However, if the Infinispan store is defined with "probuf" file then this translator works with definition objects in the protobuf file. Typical usage of HotRod protocol also dictates this requirement.

Note	**What is Infinispan** - Infinispan is a distributed in-memory key/value data store with optional schema, available under the Apache License 2.0

The following will be explained

- Usage
- Configuration of Translator
  - Defining the Metadata
  - Details on Protobuf to DDL conversion
  - Protobuf Translation Rules
- Execution Properties
- Importer Properties
- Limitations
- JCA Resource Adapter

Usage

Below is a sample VDB that can read metadata from a protobuf file based on the AddressBook quick start on http://infinispan.org site.

```
<vdb name="addressbook" version="1">
    <model name="ispn">
        <property name="importer.ProtobufName" value="addressbook.proto"/>
        <source name="localhost" translator-name="infinispan-hotrod" connection-jndi-name="java:/ispnDS"/>
        <metadata type = "NATIVE"/>
    </model>
</vdb>
```

For the above VDB to work, a connection to Infinispan is required. Below shows an example configuration for the resource-adapter that is needed. Be sure to edit the "RemoteServerList" to reflect your Infinispan server location. If you are working with "WilfFly" based Teiid installation, you need to edit the */wf-install/standalone/configuration/standalone-teiid.xml* file and add the following segment to the "resource-adapters" subsystem of the configuration.

```
<resource-adapter id="infinispanDS">
    <module slot="main" id="org.jboss.teiid.resource-adapter.infinispan.hotrod"/>
    <transaction-support>NoTransaction</transaction-support>
    <connection-definitions>
        <connection-definition class-name="org.teiid.resource.adapter.infinispan.hotrod.InfinispanManagedConnec
tionFactory"
            jndi-name="java:/ispnDS" enabled="true" use-java-context="true" pool-name="teiid-ispn-ds">
            <config-property name="RemoteServerList">
                localhost:11222
```

```
            </config-property>
        </connection-definition>
    </connection-definitions>
</resource-adapter>
```

Once you configure above resource-adapter and deploy the VDB successfully, then you can connect to the VDB using Teiid JDBC driver and issue SQL statements like

```
select * from Person;
select * PhoneNumber where number = <value>;

insert into Person (...) values (...);
update Person set name = <value> where id = <value>;
delete from person where id = <value>;
```

Configuration of Translator

Defining the Metadata

There are three different ways to define the metadata for the Infinispan model in Teiid. Choose what best fits the needs.

Metadata From New Protobuf File:

User can register a .proto file with translator configuration, which will be read in Teiid and get converted to the model's schema. Then Teiid will register this protobuf file in Infinispan. For details see Importer Properties

Example

```
<vdb name="vdbname" version="1">
    <model name="modelname">
..
        <property name="importer.ProtoFilePath" value="/path/to/myschema.proto"/>
..
    </model>
</vdb>
```

Metadata From Existing Registered Protobuf File

If the protobuf file has already been registered in your Infinispan node, Teiid can obtain it and read the protobuf directly from the cache. For details see Importer Properties

Example

```
ProtobufName
----
<vdb name="vdbname" version="1">
    <model name="modelname">
..
        <property name="importer.ProtobufName" value="existing.proto"/>
..
    </model>
</vdb>
----
```

Define Metadata in DDL

Like any other translator, you can use the <metadata> tags to define the DDL directly. For example

Example

```
    <model name="ispn">
        <source name="localhost" translator-name="infinispan-hotrod" connection-jndi-name="java:/ispnDS"/>
        <metadata type = "DDL"><![CDATA[
            CREATE FOREIGN TABLE G1 (e1 integer PRIMARY KEY, e2 varchar(25), e3 double) OPTIONS(UPDATABLE true,
 , "teiid_ispn:cache" 'g1Cache');
        ]]>
       </metadata>
       <metadata type = "NATIVE"/>
    </model>
```

Note	The "<metadata type = "NATIVE"/>" is required in order to trigger the registration of the generated protobuf file. The name of the protobuf registered in Infinispan will use the format of: schemaName + ".proto". So in the above example, it would be named ***ispn.proto***. This would be useful if another VDB wished to reference that same cache and would then use the Importer Property "importer.ProtobufName" to read it. The model must not contain dash ("-") in it's name.

For this option, a compatible protobuf definition is generated automatically during the deployment of the VDB and registered in Infinispan. Please note, if for any reason the DDL is modified (Name changed, type changed, add/remove columns) after the initial VDB is deployed, then previous version of the protobuf file and data contents need to be manually cleared before next revision of the VDB is deployed. Failure to clear will result in data encoding/corruption issues.

Details on Protobuf to DDL conversion

This section show cases an example protobuf file and shows how that file converted to relational schema in the Teiid. This below is taken from the quick start examples of Infinispan.

```
package quickstart;

/* @Indexed */
message Person {

   /* @IndexedField */
   required string name = 1;

   /* @Id @IndexedField(index=false, store=false) */
   required int32 id = 2;

   optional string email = 3;

   enum PhoneType {
      MOBILE = 0;
      HOME = 1;
      WORK = 2;
   }

   /* @Indexed */
   message PhoneNumber {

      /* @IndexedField */
      required string number = 1;

      /* @IndexedField(index=false, store=false) */
      optional PhoneType type = 2 [default = HOME];
   }

   /* @IndexedField(index=true, store=false) */
   repeated PhoneNumber phone = 4;
}
```

When Teiid's translator processes the above protobuf file, the following DDL is generated automatically for Teiid model as the relational representation.

```
CREATE FOREIGN TABLE Person (
    name string NOT NULL OPTIONS (ANNOTATION '@IndexedField', SEARCHABLE 'Searchable', NATIVE_TYPE 'string', "t
eiid_ispn:TAG" '1'),
    id integer NOT NULL OPTIONS (ANNOTATION '@Id @IndexedField(index=false, store=false)', NATIVE_TYPE 'int32',
"teiid_ispn:TAG" '2'),
    email string OPTIONS (SEARCHABLE 'Searchable', NATIVE_TYPE 'string', "teiid_ispn:TAG" '3'),
    CONSTRAINT PK_ID PRIMARY KEY(id)
) OPTIONS (ANNOTATION '@Indexed', NAMEINSOURCE 'quickstart.Person', UPDATABLE TRUE, "teiid_ispn:cache" 'personC
ache');

CREATE FOREIGN TABLE PhoneNumber (
    number string NOT NULL OPTIONS (ANNOTATION '@IndexedField', SEARCHABLE 'Searchable', NATIVE_TYPE 'string',
"teiid_ispn:TAG" '1'),
    type integer DEFAULT '1' OPTIONS (ANNOTATION '@IndexedField(index=false, store=false)', NATIVE_TYPE 'PhoneT
ype', "teiid_ispn:TAG" '2'),
    Person_id integer OPTIONS (NAMEINSOURCE 'id', SEARCHABLE 'Searchable', "teiid_ispn:PSEUDO" 'phone'),
    CONSTRAINT FK_PERSON FOREIGN KEY(Person_id) REFERENCES Person (id)
) OPTIONS (ANNOTATION '@Indexed', NAMEINSOURCE 'quickstart.Person.PhoneNumber',
    UPDATABLE TRUE, "teiid_ispn:MERGE" 'model.Person', "teiid_ispn:PARENT_COLUMN_NAME" 'phone',
    "teiid_ispn:PARENT_TAG" '4');
```

Protobuf Translation Rules

You can see from above DDL, Teiid makes use of the extension metadata properties to capture all the information required from .proto file into DDL form so that information can be used at runtime. The following are some rules the translation engine follows.

Infinispan	Mapped to Relational Entity	Example
Message	Table	Person, PhoneNumber
enum	integer attribute in table	n/a
repeated	As an array for simple types or as a separate table with one-2-many relationship to parent message.	PhoneNumber

- All required fields will be modeled as NON NULL columns
- All indexed columns will be marked as Searchable.
- The default values are captured.
- To enable updates, the top level message object MUST define @id annotation on one of its columns

Note	Notice the **@Id** annotation on the Person message's "id" attribute in protobuf file. This is **NOT** defined by Infinispan, but required by Teiid to identify the key column of the cache entry. In the absence of this annotation, only "read only" access (SELECT) is provided to top level objects. Any access to complex objects (PhoneNumber from above example) will not be provided.

IMPOTANT: When .proto file has more than single top level "message" objects to be stored as the root object in the cache, each of the objects must be stored in a different cache to avoid the key conflicts in a single cache store. This is restriction imposed by Infinispan, however Teiid's single model can have multiple of these message types. Since each of the message will be in different cache store, you can define the cache store name for the "message" object. For this, define an extension property "teiid_ispn:cache" on the corresponding Teiid's table. See below code example.

```
    <model name="ispn">
```

```
        <property name="importer.ProtobufName" value="addressbook.proto"/>
        <source name="localhost" translator-name="infinispan-hotrod" connection-jndi-name="java:/ispnDS"/>
        <metadata type = "NATIVE"/>
        <metadata type = "DDL"><![CDATA[
            ALTER FOREIGN TABLE Person OPTIONS (SET "teiid_ispn:cache" '<cache-name>');
        ]]>
       </metadata>
    </model>
```

Execution Properties

Execution properties extend/limit the functionality of the translator based on the physical source capabilities. Sometimes default properties may need to adjusted for proper execution of the translator in your environment.

Currently there are no defined execution properties for this translator.

Importer Properties

Importer properties define the behavior options of the translator during the metadata import from the physical source.

Name	Description	Default
ProtoFilePath	The file path to a Protobuf .proto file accessible to the server to be read and convert into metadata.	n/a
ProtobufName	The name of the Protobuf .proto file that has been registered with the Infinispan node, that Teiid will read and convert into metadata. The property value MUST exactly match registered name.	null

Examples

```
ProtoFilePath
----
<vdb name="vdbname" version="1">
    <model name="modelname">
..
        <property name="importer.ProtoFilePath" value="/path/to/myschema.proto"/>
..
    </model>
</vdb>
----
```

Limitations

- Bulk update support is not available.
- No transactions supported. It is currently last edit stands.
- Aggregate functions like SUM, AVG etc are not supported on inner objects (ex: PhoneNumber)
- UPSERT support on complex objects is always results in INSERT
- LOBS are not streamed, use caution as this can lead to OOM errors.

- There is no function library in Infinispan
- Array objects can not be projected currently, but they will show up in the metadata
- When using DATE/TIMESTAMP/TIME types in Teiid metadata, they are by default marshaled into a LONG type in Infinispan.
- SSL and identity support is not currently available (see TEIID-4904)

Note	**Native Queries** - Native or direct query execution is not supported through Infinispan translator.

JCA Resource Adapter

The resource adapter for this translator is a Infinispan Data Source.

JDBC Translators

The JDBC translators bridge between SQL semantic and data type differences between Teiid and a target RDBMS. Teiid has a range of specific translators that target the most popular open source and proprietary databases.

Table of Contents

- Usage
  - Type Conventions
- Execution Properties - shared by all JDBC Translators
- Importer Properties - shared by all JDBC Translators
- Native Queries
  - Direct Query Procedure
- JCA Resource Adapter

Usage

Usage of a JDBC source is straight-forward. Using Teiid SQL, the source may be queried as if the tables and procedures were local to the Teiid 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, then consider using the Translator Development to create what you need. Or log a Teiid Jira with your requirements.

Type Conventions

UID types including UUID, GUID, or UNIQUEIDENTIFIER are typically mapped to the Teiid string type. Be aware that the source will treat UID strings as non-case sensitive, but they will be in Teiid. The source may also not support the implicit conversion to the string type, thus usage in functions expecting a string value may fail at the source. Please log an issue if you encounter this situation.

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 pushdown support.	The base supported version or derived from the DatabaseMetadata.getDatabaseProductVersion string. Automatic detection requires a Connection. If there are circumstances where you are getting an exception from capabilities being unavailable (most likely due to an issue obtaining a Connection), then set DatabaseVersion property. Use the JDBCExecutionFactory.usesDatabaseVersion() method to control whether your translator requires a connection to determine capabilities.

TrimStrings	true to trim trailing whitespace from fixed length character strings. Note that Teiid 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 to indicate 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
CommentFormat	MessageFormat string to be used if UseCommentsInSourceQuery is enabled. Available properties: 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	Struct retrieval mode can be one of OBJECT - getObject value returned, COPY - returned as a `SerialStruct`, ARRAY - returned as an `Array`)	OBJECT
EnableDependentJoins	For sources that support temporary tables (DB2, Derby, H2, HSQL 2.0+, MySQL 5.0+, Oracle, PostgreSQL, SQLServer, SQP IQ, Sybase) allow dependent join pushdown	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
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. Note supported only for DB2, Oracle, PostgreSQL, SQL Server, and H2. A matching sequence will be imported to a 0-argument Teiid function name_nextval.	false
sequenceNamePattern	like pattern string to use when importing sequences. Null or % will match all.	null
useFullSchemaName	When false, directs the importer to use just the object name as the Teiid name - Note: when false importing from multiple schemas may lead to objects with duplicate names when importing from multiple schemas, which results in an exception. When true the Teiid name will be formed using the catalog and schema names as directed by the useCatalogName and useQualifiedName properties. This option does not affect the name in source property.	true
useQualifiedName	true will use name qualification for both the Teiid name and name in source as further refined by the useCatalogName and useFullSchemaName properties. Set to false to disable all qualification for both the Teiid name and the name in source, which effectively ignores the useCatalogName and useFullSchemaName properties. WARNING: when false this may lead to objects with duplicate names when importing from multiple schemas, which results in an exception.	true (rarely needs changed)
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 Teiid runtime name if applicable. false will not use the catalog name in either the name in source nor the Teiid runtime name. Only required to be set to false for sources that do not support a catalog concept, but return a non-null/non-empty catalog name in their metadata - such as HSQL.	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 Teiid 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 support returning a statistical index.	false
importStatistics	true will use database dependent logic to determine the cardinality if none is determined. Not yet supported by all database types - currently only supported by Oracle and MySQL.	false
importRowIdAsBinary	true will import RowId columns as varbinary values.	false
importLargeAsLob	true will import character and binary types larger than the Teiid 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 only import tables and views from my-schema. See also VDB Guide

```
<model ...

  <property name="importer.tableTypes" value="TABLE,VIEW"/>
  <property name="importer.schemaName" value="my-schema"/>
  ...
</model>
```

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 support 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 Teiid 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 - see Parameterizable Native Queries. The teiid_rel:non-prepared extension metadata property may be set to false to turn off parameter binding. Note this option should be used with caution as inbound may allow 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 positionally matching the result set 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 Teiid 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 security risk this exposes to execute any command against the source. To enable this feature, override the execution property called _SupportsDirectQueryProcedure to true.

By default the name of the procedure that executes the queries directly is **native**. Override the execution property _DirectQueryProcedureName to change it to another name.

The JDBC translator provides a procedure to execute any ad-hoc SQL query directly against the source without Teiid parsing or resolving. Since the metadata of this procedure's results are not known to Teiid, they are returned as an object array. ARRAYTABLE can be used construct tabular output for consumption by client applications.

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
```

JCA Resource Adapter

The resource adapter for this translator provided through data source in WildFly, See to Admin Guide section WildFly Data Sources for configuration.

Actian Vector Translator (actian-vector)

Also see common JDBC Translator Information

The Actian Vector Translator, known by the type name ***actian-vector***, is for use Actian Vector in Hadoop.

Download the JDBC driver at http://esd.actian.com/platform. Note the port number in connection URL is "AH7" which maps to 16967.

Apache HBase Translator (phoenix)

Also see common JDBC Translator 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 - see Phoenix Data Sources 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 there may be other translators introduced in the furture that also connect to HBase.

The DatabaseTimezone property should not be used with this translator.

The HBase Translator doesn’t support Joins. Phoenix uses the HBase Table Row ID as the Primary Key, which map to . This Translator is developed with Phoenix 4.3+ for HBase 0.98.1+.

Note	The translator implements INSERT/UPDATE through the Phoenix UPSERT operation. This means you can see different behavior than with standard INSERT/UPDATE - such as 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 will not look for unique constraints and defaults to not importing foreign keys.

Note	The translator supports offset 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, please 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.

Cloudera Impala Translator (impala)

Also see common JDBC Translator Information

The Coudera 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 Teiid 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 Teiid engine.

The DatabaseTimeZone translator property should not be used.

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, modeled them on 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.

Note	If 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.

Note	Some versions of Impala requires the use of a LIMIT when performing an ORDER BY. If no default is configured in Impala, then an exception can occur when a Teiid 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. See the Cloudera docs for more on limit options - such as controlling what happens when the limit is exceeded.

Note	The Impala JDBC driver seems to have issues with PreparedStatements and statement parsing in general that may require disabling useBindVariables - see link:https://issues.jboss.org/browse/TEIID-4610

DB2 Translator (db2)

Also see common JDBC Translator Information

The DB2 Translator, known by the type name ***db2***, if for use with DB2 8 or later and DB2 for i 5.4 or later.

DB2 specific 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. Defaults to true. Some older versions, or instances running in a conversion mode, of DB2 lack full common table expression support and may need support disabled.

Derby Translator (derby)

Also see common JDBC Translator Information

The Derby Translator, known by the type name ***derby***, is for use with Derby 10.1 or later.

Exasol Translator (exasol)

Also see common JDBC Translator 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 Teiid engine works in the NULL LOW mode, which means that depending on whether the ordering is pushed down to Exasol or done by the engine, you may observe NULLs either at the beginning or at the end of returned results. If you need consistency, you can run Teiid with org.teiid.pushdownDefaultNullOrder=true and enforce the NULL LOW ordering (may impact performance).

Greenplum Translator (greenplum)

Also see common JDBC Translator 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.

H2 Translator (h2)

Also see common JDBC Translator Information

The H2 Translator, known by the type name ***h2***, is for use with H2 version 1.1 or later.

Hive Translator (hive)

Also see common JDBC Translator 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 has limited support for data types. It does not have native support for time/xml or LOBs. These limitations are reflected in the translator capabilities. A Teiid 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 Teiid engine.

The DatabaseTimeZone translator property should not be used.

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, modeled them on source table, but do not include them in selection columns.

Note	The Hive importer does not have concept of catalog or source schema, nor does it import keys, procedures, indexes, etc.

Import Properties

- *trimColumnNames*- For Hive 0.11.0 and later the the DESCRIBE command metadata is inappropriately returned with padding, set to true to strip trim 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.

Note	If false the typical JDBC DatabaseMetaData calls are not used so not all of the common JDBC importer properties are applicable to Hive. You may still use excludeTables regardless.

"Database Name"

When the database name used in the Hive is different than "default", the metadata retrieval and execution of queries does not work as expected in Teiid, as 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 may configure your connection source to issue "use {database-name}".

Teiid in WildFly environment set the following in data source configuration.

```
    <new-connection-sql>use {database-name}</new-connection-sql>
```

This is fixed in > 0.13 version Hive Driver. See https://issues.apache.org/jira/browse/HIVE-4256

Limitations

Empty tables may report their description without datatype information. You will need exclude these tables or use the useDatabaseMetaData option for import.

HSQL Translator (hsql)

Also see common JDBC Translator Information

The HSQL Translator, known by the type name ***hsql***, is for use with HSQLDB 1.7 or later.

Informix Translator (informix)

Also see common JDBC Translator Information

The Informix Translator, known by the type name ***informix***, is for use with any Informix version.

Known Issues

TEIID-3808 - The Informix driver handling of timezone information is inconsistent - even if the databaseTimezone translator property is set. Consider ensuring that the Informix server and the application server are in the same timezone.

Ingres Translators (ingres / ingres93)

Also see common JDBC Translator Information

The Ingres translation is supported by 2 translators.

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.

Intersystems Cache Translator (intersystems-cache)

Also see common JDBC Translator Information

The Intersystem Cache Translator, known by the type name ***intersystems-cache***, is for use with Intersystems Cache Object database (only relational aspect of it).

JDBC ANSI Translator (jdbc-ansi)

Also see common JDBC Translator Information

The JDBC ANSI translator, known by the type name ***jdbc-ansi***, declares support for most SQL constructs supported by Teiid, except for row limit/offset and EXCEPT/INTERSECT. Translates source SQL into ANSI compliant syntax. This translator should be used when another more specific type is not available. If source exceptions arise from unsupported SQL, then consider using the JDBC Simple Translator to further restrict capabilities, or create a Custom Translator / create an enhancement request.

JDBC Simple Translator (jdbc-simple)

Also see common JDBC Translator Information

The JDBC Simple translator, known by the type name ***jdbc-simple***, is the same as jdbc-ansi, except disables support for nearly all pushdown constructs for maximum compatibility.

MetaMatrix Translator (metamatrix)

Also see common JDBC Translator Information

The MetaMatrix Translator, known by the type name ***metamatrix***, is for use with MetaMatrix 5.5.0 or later.

Microsoft Access Translators

Also see common JDBC Translator 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 Teiid connection importer the importer defaults to importKeys=false and excludeTables=.**[.]MSys.** to avoid issues with the metadata provided by the JDBC ODBC bridge. You may 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 for the UCanAccess driver.

Microsoft SQL Server Translator (sqlserver)

Also see common JDBC Translator 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 compatible e.g. JTDS 1.2 or later) should be used. The SQL Server DatabaseVersion property may be set to 2000, 2005, 2008, or 2012, but otherwise expects a standard version number - e.g. "10.0".

Sequence Support

With Teiid 8.5+ sequence operations may be modeled as source functions.

With Teiid 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*- indicates that the open source JTDS driver is being used. Defaults to false.

ModeShape Translator (modeshape)

Also see common JDBC Translator Information

The ModeShape Translator, known by the type name ***modeshape***, is for use with Modeshape 2.2.1 or later.

Usage

The PATH, NAME, LOCALNODENAME, DEPTH, and SCORE functions should be accessed as pseudo-columns, e.g. "nt:base"."jcr:path".

Teiid UFDs (prefixed by JCR_) are available for CONTIANS, ISCHILDNODE, ISDESCENDENT, ISSAMENODE, REFERENCE - see the JCRFunctions.xmi. If a selector name is needed in a JCR function, you should use the pseudo-column "jcr:path", e.g. JCR_ISCHILDNODE(foo.jcr_path, 'x/y') would become ISCHILDNODE(foo, `x/y') in the ModeShape query.

An additional pseudo-column "mode:properties" should be imported by setting the ModeShape JDBC connection property teiidsupport=true. The column "mode:properties" should be used by the JCR_REFERENCE and other functions that expect a .* selector name, e.g. JCR_REFERENCE(nt_base.jcr_properties) would become REFERENCE("nt:base".*) in the ModeShape query.

MySQL Translator (mysql/mysql5)

Also see common JDBC Translator Information

MySQL/MariaDB translation is supported by 2 translators.

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.

Also supports compatible MySQL derivatives including MariaDB.

Usage

The MySQL Translators expect the database or session to be using ANSI mode. If the database is not using ANSI mode, an initialization query should be used on the pool to set ANSI mode:

```
set SESSION sql_mode = 'ANSI'
```

If you may deal with null timestamp values, then set the connection property zeroDateTimeBehavior=convertToNull. Otherwise you'll get conversion errors in Teiid that `0000-00-00 00:00:00' cannot be converted to a timestamp.

Warning	If retrieving large result sets, you should consider setting the connection property useCursorFetch=true, otherwise MySQL will fully fetch result sets into memory on the Teiid 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, you should change affected source BOOLEAN columns to have a native type of "TINYINT(1)" rather than BIT so that the translator can appropriately handle the boolean conversion.

Netezza Translator (netezza)

Also see common JDBC Translator Information

The Netezza Translator, known by the type name ***netezza***, is for use with any Netezza version.

Usage

The current Netezza vendor supplied JDBC driver performs poorly with single transactional updates. As is generally the case when possible use batched updates.

Execution Properties

Netezza specific execution properties:

- *SqlExtensionsInstalled*- indicates that SQL Extensions, including support for REGEXP_LIKE, are installed. All other REGEXP functions are then available as pushdown functions. Defaults to false.

Oracle Translator (oracle)

Also see common JDBC Translator Information

The Oracle Translator, known by the type name ***oracle***, is for use with Oracle 9i or later.

Note	The Oracle provide JDBC driver may cause memory issues due to excessive buffer usage. Please see a related issue and an Oracle whitepaper.

Importer Properties

- *useGeometryType*- Use the Teiid 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

Sequences may be used with the Oracle translator. A sequence may be modeled as a table with a name in source of DUAL and columns with the name in source set to <sequence name>.[nextval|currval]

With Teiid 10.0+ sequences may be imported automatically import properties.

Teiid 8.4 and Prior Oracle Sequence DDL

```
CREATE FOREIGN TABLE seq (nextval integer OPTIONS (NAMEINSOURCE 'seq.nextval'), currval integer options (NAMEIN
SOURCE 'seq.currval') ) OPTIONS (NAMEINSOURCE 'DUAL')
```

With Teiid 8.5 it's no longer necessary to rely on a table representation and Oracle specific handling for sequences. See DDL Metadata for representing currval and nextval as source functions.

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 the name in source to `<element name>:SEQUENCE=<sequence name>.<sequence value>`.

Rownum

A rownum column can also added to any Oracle physical table to support 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)
```

Geo Spatial function support

Oracle translator supports geo spatial functions. The supported functions are:

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

The Oracle translator, depending upon the version of Oracle, will register other non-geospatial pushdown functions with the engine. This includes:

- TRUNC, both numeric and timestamp versions
- LISTAGG, which requires the Teiid 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 need to use client driver later than 11.2, have the xdb.jar and xmlparserv2.jar jars in the classpath, and set the system property oracle.jdbc.getObjectReturnsXMLType="false". See also the Oracle documentation.

OSISoft PI Translator (osisoft-pi)

Also see common JDBC Translator Information

The OSISoft Translator, known by the type name ***osisoft-pi***, is for use with OSIsoft PI OLEDB Enterprise. This translator uses the JDBC driver provided by the OSISoft. The driver is not provided with Teiid install, this needs be downloaded from OSISoft and installed correctly on Teiid server according to OSISoft documentation *PI-JDBC-2016-Administrator-Guide.pdf* or latest document.

Install on Linux

Make sure you have OpenSSL libraries installed, and you have following "export" added correctly in your shell environment variables. Otherwise you can also add in *<WildFly>/bin/standalone.sh* file or *<WildFly>/bin/domain.sh* file.

```
export PI_RDSA_LIB=/<path>/pipc/jdbc/lib/libRdsaWrapper-1.5b.so
export PI_RDSA_LIB64=/<path>/pipc/jdbc/lib/libRdsaWrapper64-1.5b.so
```

Please also note to execute from Linux, you also need install 'gSoap' library, as PI JDBC driver uses SOAP over HTTPS to communicate with PI server.

Install on Windows

Follow the installation program provided by OSISoft for installing the JDBC drivers. Make sure you have the following environment variables configured.

```
PI_RDSA_LIB     C:\Program Files (x86)\PIPC\JDBC\RDSAWrapper.dll
PI_RDSA_LIB64   C:\Program Files\PIPC\JDBC\RDSAWrapper64.dll
```

Installing the JDBC driver for Teiid (same for both Linux and Windows)

Then copy the module directory from *<WildFly>/teiid/datasources/osisoft-pi/modules directory into _<WilfFly>/modules* directory. Then find the "PIJDBCDriver.jar" file from the installation directory, and copy it to _<WildFly>/module/system/layers/dv/com/osisoft/main" directory. Then add the driver definition to the standalone.xml file by editing the file and adding something similar to below

```
    <drivers>
       <driver name="osisoft-pi" module="com.osisoft">
           <driver-class>com.osisoft.jdbc.Driver</driver-class>
       </driver>
    </drivers>
```

That completes the configuration of the PI driver in the Teiid. We still have not created a connection to the PI server. you can start the server now.

Creating a Data Source to PI

You can execute following similar CLI script to create a datasource

```
/subsystem=datasources/data-source=pi-ds:add(jndi-name=java:/pi-ds,  driver-
name=osisoft-pi, connection-url=jdbc:pioledbent://<DAC Server>/Data Source=<AF
Server>; Integrated Security=SSPI,user-name=user, password=mypass)
/subsystem=datasources/data-source=pi-ds:enable
```

this will create following XML in standalone.xml or domain.xml (you can also directly edit these files and add manually)

```
<datasource jndi-name="java:/pi-ds" pool-name="pi-ds">
    <connection-url>jdbc:pioledbent://<DAC Server>/Data Source=<AF Server>;
Integrated Security=SSPI</connection-url>
    <driver>osisoft-pi</driver>
    <pool>
        <prefill>false</prefill>
        <use-strict-min>false</use-strict-min>
        <flush-strategy>FailingConnectionOnly</flush-strategy>
    </pool>
    <security>
        <user-name>user</user-name>
        <password>mypass</password>
    </security>
</datasource>
```

Now you have fully configured the Teiid with PI database connection. You can create VDB that can use this connection to issue the queries.

Usage

You can develop a VDB like follows to fetch metadata from PI and give you access to executing queries against PI.

pi-vdb.xml

```
<vdb name="pi" version="1">
    <model name="AF">
        <property name="importer.importProcedures" value="true"/>
        <source connection-jndi-name="java:/pi-ds" name="pi-connector" translator-
name="osisoft-pi"/>
    </model>
</vdb>
```

Deploy this file into Teiid using CLI or using management console

```
deploy pi-vdb.xml
```

Once the metadata is loaded and VDB is active you can use Teiid JDBC/ODBC driver or OData to connect to the VDB and issue queries.

PI Translator Capabilities

PI translator is extension of *jdbc-ansi* translator, so all the SQL ANSI queries are supported. PI translator also supports LATERAL join with Table Valued Functions (TVF). An example Teiid query looks like

```
SELECT EH.Name, BT."Time", BT."Number of Computers", BT."Temperature"
    FROM Sample.Asset.ElementHierarchy EH
    LEFT JOIN  LATERAL (exec "TransposeArchive_Building Template"(EH.ElementID,
TIMESTAMPADD(SQL_TSI_HOUR, -1, now()), now())) BT on 1=1
    WHERE EH.ElementID IN (SELECT ElementID FROM Sample.Asset.ElementHierarchy
WHERE Path='\Data Center\')
```

Note	ANSI SQL semantics require a ON clause, but CROSS APPLY or OUTER APPLY do no have a ON clause, so for this reason user need to pass in a dummy ON clause like ON (1 = 1), which will be ignored when converted to APPLY clause which will be pushed down.

By default this translator turns off the "importer.ImportKeys" to false.

Note	The PI data type, "GUID" will need to be modeled as "String" and must define the NATIVE_TYPE on column as "guid", then Teiid translator will appropriately convert the data back forth with the PI datasource’s native guid type with appropriate type casting from string.

Pushdown Functions

PI accepts time interval literals that are not recognized by Teiid. If you wish to make a comparison based upon an interval, use the PI.inteveral function:

```
select * from Archive a where a.time between PI.interval('*-14d') and
PI.interval('*')
```

Known Issues: TEIID-5123 - Casting a string containing a negative or zero value (e.g. '-24' or '0') to Float/Single fails with PI Jdbc driver.

PostgreSQL Translator (postgresql)

Also see common JDBC Translator Information

The PostgreSQL Translator, known by the type name ***postgresql***, is for use with 8.0 or later clients and 7.1 or later server.

Execution Properties

PostgreSQL specific execution properties:

- *PostGisVersion*- indicate the PostGIS version in use. Defaults to 0 meaning PostGIS is not installed. Will be set automatically if the database version is not set.
- *ProjSupported*- boolean indicating if Proj is support for PostGis. 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. Later versions of Teiid will omit these tables automatically. Older versions of Teiid may need the importer.tableType property or other filtering set.

PrestoDB Translator (prestodb)

Also see common JDBC Translator Information

The PrestoDB translator, known by the type name ***prestodb***, exposes querying functionality to PrestoDB Data Sources. In data integration respect, PrestoDB has very similar capabilities of Teiid, however it goes beyond in terms of distributed query execution with multiple worker nodes. Teiid's execution model is limited to single execution node and focuses more on pushing the query down to sources. Currently Teiid has much more complete query support and many enterprise features.

Capabilities

The PrestoDB translator supports only SELECT statements with a restrictive set of capabilities. This translator is developed with 0.85 version of PrestoDB and capabilities are designed for this version. With new versions of PrestoDB Teiid will adjust the capabilities of this translator. Since PrestoDB exposes a relational model, the usage of this is no different than any RDBMS source like Oracle, DB2 etc. For configuring the PrestoDB consult the PrestoDB documentation.

Tip	PrestoDB not support multiple columns in the ORDER BY with in JOIN situations well, the translator property `supportsOrderBy` can use to disable Order by in some specific situations.

Tip	Some versions of PrestoDB not support null as valid values in subquery well, to check the PrestoDB whether support null as valid values in subquery if you hit releated error.

Tip	PrestoDB not support transaction, define a no-tx-datasource is recommend.

Note	Every catalogs in PrestoDB has a `information_schema` schema by default, If configure multiple catalogs, it should be consider to use import options to filter the schemas/tables, to avoid `Duplicate Table` error cause VDB deploy failed. For instance, set `catalog` to a specific catalog name to match the catalog name as it is stored in the prestodb, set `schemaName` to a single schema, or set `excludeTables` to a regular expression to filter tables by matching result.

Note	PrestoDB JDBC driver uses Joda-Time library to work with time/date/timestamp. If you need to customize server's time zone (e.g. setting -Duser.timezone via JAVA_OPTS), you cannot use `GMT/…` ID as Joda-Time does not regonize it. However, you can use equivalent `Etc/…` ID. For more details see Joda-Time timezones.

Redshift Translator (redshift)

Also see common JDBC Translator Information

The Redshift Translator, known by the type name ***redshift***, is for use with the Redshift database. This translator is an extension of the PostgreSQL Translator and inherits its options.

SAP Hana Translator (hana)

Also see common JDBC Translator 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 Teiid substring function when the from index exceeds the length of the string. SAP Hana will return an empty string, while Teiid produces a null value.

SAP IQ Translator (sap-iq)

Also see common JDBC Translator Information

The SAP IQ Translator, known by the type name ***sap-iq***, is for use with Sybase/SAP IQ version 15.1 or later. The translator name ***sybaseiq*** has been deprecated.

Sybase Translator (sybase)

The Sybase Translator, known by the type name ***sybase***, is for use with Sybase version 12.5 or later.

If using the the default native import and no import properties are specified (not recommended, see import properties below), then exceptions can be thrown retrieving system table information. You should specify a schemaName, or schemaPattern, or use excludeTables to exclude system tables if this occurs.

If the name in source metadata contains quoted identifiers (such as required by reserved words or words containing 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**:

Driver URL with SQLINITSTRING

```
jdbc:sybase:Tds:host.at.some.domain:5000/db_name?SQLINITSTRING=set quoted_identifier on
```

If you are a jconnect Sybase driver and will target the source for dependent joins, you should allow the translator to send more values by setting the JCONNECT_VERSION. Otherwise you will get exceptions with statements that have more than 481 bind values:

Driver URL with JCONNECT_VERSION

```
jdbc:sybase:Tds:host.at.some.domain:5000/db_name?SQLINITSTRING=set quoted_identifier on&JCONNECT_VERSION=6
```

Sybase specific execution properties:

- *JtdsDriver*- indicates that the open source JTDS driver is being used. Defaults to false.

Teiid Translator (teiid)

Also see common JDBC Translator Information

The Teiid Translator, known by the type name ***teiid***, is for use with Teiid 6.0 or later.

Teradata Translator (teradata)

Also see common JDBC Translator Information

The Teradata Translator, known by the type name ***teradata***, is for use with Teradata V2R5.1 or later.

With Teradata driver version 15 date, time, and timestamp values by default will be adjusted for the Teiid server timezone. To remove this adjustment, set the translator DatabaseTimezone property to GMT or whatever the Teradata server defaults to.

Vertica Translator (vertica)

Also see common JDBC Translator Information

The Vertica Translator, known by the type name ***vertica***, is for use with Vertica 6 or later.

JPA Translator

The JPA translator, known by the type name *jpa2*, can reverse a JPA object model into a relational model, which can then be integrated with other relational or non-relational sources.

For information on JPA persistence in a WildFly, see JPA Reference Guide.

Properties

The JPA Translator currently has no import or execution properties.

Native Queries

JPA source procedures may be created using the teiid_rel:native-query extension - see Parameterizable Native Queries. 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. See the query syntax below.

Direct Query Procedure

Note	This feature is turned off by default because of the security risk this exposes to execute any command against the source. To enable this feature, override the execution property called _SupportsDirectQueryProcedure to true.

Tip	By default the name of the procedure that executes the queries directly is **native**. Override the execution property _DirectQueryProcedureName to change it to another name.

The JPA translator provides a procedure to execute any ad-hoc JPA-QL query directly against the source without Teiid parsing or resolving. Since the metadata of this procedure's results are not known to Teiid, they are returned as object array. User can use ARRAYTABLE can be used construct tabular output for consumption by client applications. Teiid exposes this procedure with a simple query structure as below

Select

Select Example

```
SELECT x.* FROM (call jpa_source.native('search;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 - see Parameterizable Native Queries to substitute parameter values.

Delete

Delete Example

```
SELECT x.* FROM (call jpa_source.native('delete;<jpa-ql>')) w,
 ARRAYTABLE(w.tuple COLUMNS "updatecount" integer) AS x
```

form the above code, the "delete" keyword followed by JPA-QL for delete operation.

Update

Create Example

```
SELECT x.* FROM
 (call jpa_source.native('update;<jpa-ql>')) w,
 ARRAYTABLE(w.tuple COLUMNS "update_count" integer) AS x
```

form the above code, the "update" keyword must be followed by JPA-QL for the update statement.

Create

Update Example

```
SELECT x.* FROM
 (call jpa_source.native('create;', <entity>)) w,
 ARRAYTABLE(w.tuple COLUMNS "update_count" integer) AS x
```

Create operation needs to send "create" word as marker and send the entity as a the first parameter.

LDAP Translator

The LDAP translator is implemented by the org.teiid.translator.ldap.LDAPExecutionFactory class and known by the translator type name ldap. The LDAP translator exposes an LDAP directory tree relationally with pushdown support for filtering via criteria. This is typically coupled with the LDAP resource adapter.

Note	The resource adapter for this translator is provided by configuring the ldap data source in the JBoss EAP instance.

Execution Properties

Name	Description	Default
SearchDefaultBaseDN	Default Base DN for LDAP Searches	null
SearchDefaultScope	Default Scope for LDAP Searches. Can be one of SUBTREE_SCOPE, OBJECT_SCOPE, ONELEVEL_SCOPE.	ONELEVEL_SCOPE
RestrictToObjectClass	Restrict Searches to objectClass named in the Name field for a table	false
UsePagination	Use a PagedResultsControl to page through large results. This is not supported by all directory servers.	false
ExceptionOnSizeLimitExceeded	Set to true to throw an exception when a SizeLimitExceededException is received and a LIMIT is not properly enforced.	false

There are no import settings for the ldap translator; it also does not provide metadata.

Metadata Options

SEARCHABLE 'equality_only'

For openldap, apacheds, and other ldap servers dn attributes have search restrictions, such that only equality predicates are supported. Use SEARCHABLE `equality_only` to indicates that only equality predicates should be pushed down. Any other predicate would need evaluated in the engine. For example

```
col string OPTIONS (SEARCHABLE 'equality_only', ...)
```

teiid_ldap:rdn_type

Used on a column with a dn value to indicate the rdn to extract. If the entry suffix does not match this rdn type, then no row will be produced. For example

```
col string OPTIONS ("teiid_ldap:rdn_type" 'cn', ...)
```

teiid_ldap:dn_prefix

Used on a column if rdn_type is specified to indicates that the values should match this prefix, no row will be produced for a non-matching entry. For example

```
col string OPTIONS ("teiid_ldap:rdn_type" 'cn', "teiid_ldap:dn_prefix" 'ou=groups,dc=example,dc=com', ...)
```

Multivalued Attribute Support

If one of the methods below is not used and the attribute is mapped to a non-array type, then any value may be returned on a read operation. Also insert/update/delete support will not be multi-value aware.

Concatenation

String columns with a default value of "multivalued-concat" will concatenate all attribute values together in alphabetical order using a ? delimiter. Insert/update will function as expected if all applicable values are supplied in the concatenated format.

Array support

Multiple attribute values may also supported as an array type. The array type mapping also allows for insert/update operations.

For example here is ddl with objectClass and uniqueMember as arrays:

```
create foreign table ldap_groups (objectClass string[], DN string, name string options (nameinsource 'cn'), uni
queMember string[]) options (nameinsource 'ou=groups,dc=teiid,dc=org', updatable true)
```

The array values can be retrieved with a SELECT. An example insert with array values could look like:

```
insert into ldap_groups (objectClass, DN, name, uniqueMember) values (('top', 'groupOfUniqueNames'), 'cn=a,ou=g
roups,dc=teiid,dc=org', 'a', ('cn=Sam Smith,ou=people,dc=teiid,dc=org',))
```

Unwrap

When a multivalued attribute represents an association between entities, it’s possible to use extension metadata properties to represent it as a 1-to-many or many-to-many relationship.

Example many-to-many DDL:

```
CREATE foreign table users (username string primary key options (nameinsource 'cn'), surname string options (na
meinsource 'sn'), ...) options (nameinsource 'ou=users,dc=example,dc=com');

CREATE foreign table groups (groupname string primary key options (nameinsource 'cn'), description string, ...)
 options (nameinsource 'ou=groups,dc=example,dc=com');

CREATE foreign table membership (username string options (nameinsource 'cn'), groupname options (nameinsource '
memberOf', SEARCHABLE 'equality_only', "teiid_rel:partial_filter" true, "teiid_ldap:unwrap" true, "teiid_ldap:d
n_prefix" 'ou=groups,dc=example,dc=com', "teiid_ldap:rdn_type" 'cn'), foreign key (username) references users (
username), foreign key (groupname) references groups (groupname)) options (nameinsource 'ou=users,dc=example,dc
=com');
```

The result from "select * from membership" will then produce 1 row for each memberOf and the key value will be based upon the cn rdn value rather than the full dn. Also queries that join between users and membership will be pushed as a single query.

If the unwrap attribute is missing or there are no values, then a single row with a null value will be produced.

Native Queries

LDAP procedures may optionally have native queries associated with them - see Parameterizable Native Queries. The operation prefix (select;, insert;, update;, delete; - see below for more) must be present in the native-query, butit will not be issued as part of the query to the

Example DDL for an LDAP native procedure

```
CREATE FOREIGN PROCEDURE proc (arg1 integer, arg2 string) OPTIONS ("teiid_rel:native-query" 'search;context-nam
e=corporate;filter=(&(objectCategory=person)(objectClass=user)(!cn=$2));count-limit=5;timeout=$1;search-scope=O
NELEVEL_SCOPE;attributes=uid,cn') returns (col1 string, col2 string);
```

Parameter values will have reserved characters escaped, but are otherwise directly substituted into the query.

Direct Query Procedure

Note	This feature is turned off by default because of the security risk this exposes to execute any command against the source. To enable this feature, override the execution property called _SupportsDirectQueryProcedure to true.

Tip	By default the name of the procedure that executes the queries directly is **native**. Override the execution property _DirectQueryProcedureName to change it to another name.

The LDAP translator provides a procedure to execute any ad-hoc LDAP query directly against the source without Teiid parsing or resolving. Since the metadata of this procedure's results are not known to Teiid, they are returned as an object array. ARRAYTABLE can be used construct tabular output for consumption by client applications.

Search

Search Example

```
SELECT x.* FROM (call pm1.native('search;context-name=corporate;filter=(objectClass=*);count-limit=5;timeout=6;
search-scope=ONELEVEL_SCOPE;attributes=uid,cn')) w,
 ARRAYTABLE(w.tuple COLUMNS "uid" string , "cn" string) AS x
```

from the above code, the "**search**" keyword followed by below properties. Each property must be delimited by semi-colon (;) If a property contains a semi-colon (;), it should be escaped by another semi-colon - see alsoParameterizable Native Queries and the native-query procedure example above.

Name	Description	Required
context-name	LDAP Context name	Yes
filter	query to filter the records in the context	No
count-limit	limit the number of results. same as using LIMIT	No
timeout	Time out the query if not finished in given milliseconds	No
search-scope	LDAP search scope, one of SUBTREE_SCOPE, OBJECT_SCOPE, ONELEVEL_SCOPE	No
attributes	attributes to retrieve	Yes

Delete

Delete Example

```
SELECT x.* FROM (call pm1.native('delete;uid=doe,ou=people,o=teiid.org')) w,
 ARRAYTABLE(w.tuple COLUMNS "updatecount" integer) AS x
```

form the above code, the "**delete**" keyword followed the "DN" string. All the string contents after the "delete;" used as DN.

Create or Update

Create Example

```
SELECT x.* FROM
 (call pm1.native('create;uid=doe,ou=people,o=teiid.org;attributes=one,two,three', 'one', 2, 3.0)) w,
 ARRAYTABLE(w.tuple COLUMNS "update_count" integer) AS x
```

form the above code, the "**create**" keyword followed the "DN" string. All the string contents after the "create;" is used as DN. It also takes one property called "attributes" which is comma separated list of attributes. The values for each attribute is specified as separate argument to the "native" procedure.

Update is similar to "create".

Update Example

```
SELECT x.* FROM
 (call pm1.native('update;uid=doe,ou=people,o=teiid.org;attributes=one,two,three', 'one', 2, 3.0)) w,
 ARRAYTABLE(w.tuple COLUMNS "update_count" integer) AS x
```

LDAP Connector Capabilities Support

LDAP does not provide the same set of functionality as a relational database. The LDAP Connector supports many standard SQL constructs, and performs the job of translating those constructs into an equivalent LDAP search statement. For example, the SQL statement:

```
SELECT firstname, lastname, guid
FROM public_views.people
WHERE
(lastname='Jones' and firstname IN ('Michael', 'John'))
OR
guid > 600000
```

uses a number of SQL constructs, including:

- SELECT clause support
- select individual element support (firstname, lastname, guid)
- FROM support
- WHERE clause criteria support
- nested criteria support
- AND, OR support
- Compare criteria (Greater-than) support
- IN support

The LDAP Connector executes LDAP searches by pushing down the equivalent LDAP search filter whenever possible, based on the supported capabilities. Teiid automatically provides additional database functionality when the LDAP Connector does not explicitly provide support for a given SQL construct. In these cases, the SQL construct cannot be pushed down to the data source, so it will be evaluated in Teiid, in order to ensure that the operation is performed. In cases where certain SQL capabilities cannot be pushed down to LDAP, Teiid pushes down the capabilities that are supported, and fetches a set of data from LDAP. Teiid then evaluates the additional capabilities, creating a subset of the original data set. Finally, Teiid will pass the result to the client. It is useful to be aware of unsupported capabilities, in order to avoid fetching large data sets from LDAP when possible.

LDAP Connector Capabilities Support List

The following capabilities are supported in the LDAP Connector, and will be evaluated by LDAP:

- SELECT queries
- SELECT element pushdown (for example, individual attribute selection)
- AND criteria
- Compare criteria (e.g. <, ⇐, >, >=, =, !=)
- IN criteria
- LIKE criteria.
- OR criteria
- INSERT, UPDATE, DELETE statements (must meet Modeling requirements)

Due to the nature of the LDAP source, the following capability is not supported:

- SELECT queries

The following capabilities are not supported in the LDAP Connector, and will be evaluated by Teiid after data is fetched by the connector:

- Functions
- Aggregates
- BETWEEN Criteria
- Case Expressions
- Aliased Groups
- Correlated Subqueries
- EXISTS Criteria
- Joins
- Inline views
- IS NULL criteria
- NOT criteria
- ORDER BY
- Quantified compare criteria
- Row Offset
- Searched Case Expressions

- Select Distinct
- Select Literals
- UNION
- XA Transactions

Usage

ldap-as-a-datasource quickstart demonstrates using the ldap Translator to access data in OpenLDAP Server. The name of the translator to use in vdb.xml is "translator-ldap", for example:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<vdb name="ldapVDB" version="1">
<model name="HRModel">
<source name="local" translator-name="translator-ldap"
connection-jndi-name="java:/ldapDS"/>
</model>
</vdb>
```

The translator does not provide a connection to the OpenLDAP. For that purpose, Teiid has a JCA adapter that provides a connection to OpenLDAP using the Java Naming API. To define such connector, use the following XML fragment in standalone-teiid.xml. See a example in "<jboss-as>/docs/teiid/datasources/ldap"

```
<resource-adapter id="ldapQS">
<module slot="main" id="org.jboss.teiid.resource-adapter.ldap"/>
<connection-definitions>
<connection-definition
class-name="org.teiid.resource.adapter.ldap.LDAPManagedConnectionFactory"
jndi-name="java:/ldapDS" enabled="true" use-java-context="true"
pool-name="ldapDS">
<config-property name="LdapAdminUserPassword">
redhat
</config-property>
<config-property name="LdapAdminUserDN">
cn=Manager,dc=example,dc=com
</config-property>
<config-property name="LdapUrl">
ldap://localhost:389
</config-property>
</connection-definition>
</connection-definitions>
</resource-adapter>
```

The above defines the translator and connector. For more ways to create the connector see LDAP Data Sources, LDAP translator can derive the metadata based on existing Users/Groups in LDAP Server, user need to define the metadata. For example, you can define a schema using DDL:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<vdb name="ldapVDB" version="1">
<model name="HRModel">
<metadata type="DDL"><![CDATA[
CREATE FOREIGN TABLE HR_Group (
DN string options (nameinsource `dn'),
SN string options (nameinsource `sn'),
UID string options (nameinsource `uid'),
MAIL string options (nameinsource `mail'),
NAME string options (nameinsource `cn')
) OPTIONS(nameinsource `ou=HR,dc=example,dc=com', updatable true);
]]>
```

```
</metadata>
</model>
</vdb>
```

when SELECT operation below executed against table using Teiid will retrieve Users/Groups in LDAP Server:

```
SELECT * FROM HR_Group
```

LDAP Attribute Datatype Support

LDAP providers currently return attribute value types of java.lang.String and byte[], and do not support the ability to return any other attribute value type. The LDAP Connector currently supports attribute value types of java.lang.String, Timestamp, byte[], and arrays of those values. Conversion functions that are available in Teiid allow you to use models that convert a String value from LDAP into a different data type. Some conversions may be applied implicitly, and do not require the use of any conversion functions. Other conversions must be applied explicitly, via the use of CONVERT functions. Since the CONVERT functions are not supported by the underlying LDAP system, they will be evaluated in Teiid. Therefore, if any criteria is evaluated against a converted datatype, that evaluation cannot be pushed to the data source.

When converting from String to other types, be aware that criteria against that new data type will not be pushed down to the LDAP data source. This may decrease performance for certain queries.

As an alternative, the data type can remain a string and the client application can make the conversion, or the client application can circumvent any LDAP supports ⇐ and >=, but has no equivalent for < or >. In order to support < or > pushdown to the source, the LDAP Connector will translate < to ⇐, and it will translate > to >=. When using the LDAP Connector, be aware that strictly-less-than and strictly-greater-than comparisons will behave differently than expected. It is advisable to use ⇐ and >= for queries against an LDAP based data source, since this has a direct mapping to comparison operators in LDAP.

LDAP: Testing Your Connector

You must define LDAP Connector properties accurately or the Teiid server will return unexpected results, or none at all.

LDAP: Console Deployment Issues

The Console shows an Exception That Says Error Synchronizing the Server, If you receive an exception when you synchronize the server and your LDAP Connector is the only service that does not start, it means that there was a problem starting the connector. Verify whether you have correctly typed in your connector properties to resolve this issue.

JCA Resource Adapter

The resource adapter for this translator provided through "LDAP Data Source", Refer to Admin Guide for configuration.

Loopback Translator

The Loopback translator, known by the type name *loopback*, provides a quick testing solution. It supports all SQL constructs and returns default results, with some configurable behavior.

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
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 to mimic the capabilities of	

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 may not be useful for your situation). To enable this behavior, set the DelegateName property to the name of the translator you wish to mimic. For example to disable all capabilities, set the DelegateName property to "jdbc-simple".

JCA Resource Adapter

A source connection is not required for this translator.

Microsoft Excel Translator

The Microsoft Excel Translator, known by the type name *excel*, exposes querying functionality to Excel documents using File Data Sources. Microsoft Excel is a popular spreadsheet software that is used by all the organizations across the globe for simple reporting purposes. This translator provides an easy way read a Excel spreadsheet and provide contents of the spreadsheet in the tabular form that can be integrated with other sources in Teiid.

Note	**"Does it only work on Windows?"** - No, it works on all platforms, including Windows and Linux. This translator uses Apache POI libraries to access the Excel documents which are platform independent.

Usage

The below 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

Excel translator supports "source metadata" feature, where given Excel workbook, it can introspect and build the schema based on the Sheets defined inside it. There are options available for you guide, to be able to detect header columns and data columns in a work sheet to define the correct metadata of a table.

VDB Example

The below shows an example of a VDB, that shows a exposing a Excel Document.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<vdb name="excelvdb" version="1">
    <model name="excel">
        <property name="importer.headerRowNumber" value="1"/>
        <property name="importer.ExcelFileName" value="names.xls"/>
        <source name="connector" translator-name="excel"  connection-jndi-name="java:/fileDS"/>
    </model>
</vdb>
```

"connection-jndi-name" in above represents connection to Excel document. The Excel translator does NOT provide a connection to the Excel Document. For that purpose, Teiid uses File JCA adapter that provides a connection to Excel. To define such connector, see File Data Sources or see an example in "<jboss-as>/docs/teiid/datasources/file". Once you configure both of the above, you can deploy them to Teiid Server and access the Excel Document using JDBC/ODBC/OData protocol.

Note	**"Headers in Document?"** - If you have headers in the Excel document, you can guide the import process to select the cell headers as the column names in the table creation process. See "Import Properties" section below on defining the "import" properties.

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 pattern to read data from multiple Excel documents in the same directory, in single file case choose 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

It is highly recommended that you define all the above importer properties, such that information inside the Excel Document is correctly interpreted.

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 cell must be a string value which can be forced by putting a single quote ' in front of the numeric text of the cell, or by putting a single space in front of the numeric text.

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

Metadata Extension Properties are the properties that are defined on the schema artifacts like Table, Column, Procedure etc, to describe how the translator needs to interact or interpret with source systems. All the properties are defined with 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 below shows an example table that is defined using the Extension Metadata Properties.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<vdb name="excelvdb" version="1">
    <model name="excel">
        <source name="connector" translator-name="excel"  connection-jndi-name="java:/fileDS"/>
         <metadata type="DDL"><![CDATA[
             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_NU
MBER" '2')
         ]]> </metadata>
    </model>
</vdb>
```

Note	**"Extended capabilities using ROW_ID column"** - If you define 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 can **not** be used as predicates in a query.

Tip	User does not have to depend upon "source metadata" import to create the schema represented by Excel document, they can manually create a source table and add the appropriate extension properties to make a fully functional model. If you introspect the schema model created by the import, it would look like above.

With 10.3+ the Excel translator does support updates with a couple of limitations: * The ROW_ID can not be directly modified or used as an insert value. * Update and insert values must be literals. * Updates are not transactional - the write lock is only held while writing the file and not over the entire update, thus it is possible for one update to overwrite another.

The ROW_ID of an inserted row can be returned as a generated key.

JCA Resource Adapter

The Teiid specific Excel Resource Adapter does not exist, user should use File JCA adapter with this translator. See File Data Sources for opening a File based connection.

Native Queries

Note	This feature is not applicable for Excel translator.

Direct Query Procedure

Note	This feature is not applicable for Excel translator.

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 Teiid SQL queries into MongoDB based queries. It supports 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 a MongoDB kind of NOSQL store for scalability and performance. For example, applications like storing audit logs or managing web site data fits well with MongoDB, and does not require using a structural database like Oracle, Postgres etc. 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 it co-locates the related information to achieve de-normalization that typically requires either duplication of data or joins to achieve querying in a relational database.

To make MongoDB work with Teiid the challenge for the MongoDB translator is "How to design a MongoDB store that can achieve the balance between relational and document based storage?" In our opinion the advantages of "schema-less" design are great at development time, not much at runtime except in few special situations. "Schema-less" can also be a problem with migration of application versions and the ability to query and make use of returned information effectively.

Since it is hard and may be impossible in certain situations to derive a schema based on existing the MongoDB collection(s), Teiid approaches the problem in reverse compared to other translators. When working with MongoDB, Teiid requires the user to define the MongoDB schema upfront using Teiid metadata. Since Teiid only allows relational schema as its metadata, the user needs to define their MongoDB schema in relational terms using tables, procedures, and functions. For the purposes of MongoDB, the Teiid metadata has been extended to support extension properties that can be defined on the table to convert it into a MongoDB based document. These extension properties let users define, how a MongoDB document is structured and stored. Based on the relationships (primary-key, foreign-key) defined on a table, and the 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 (see the de-normalization comment above). Thus a relational schema based design, but document based storage in MongoDB.

Table of Contents

- Who is the primary audience for this translator?
- Usage
- Data Types
- Importer Properties
- Extension Metadata Properties To Build Complex Documents
  - ONE-2-ONE Mapping
  - ONE-2-MANY Mapping.
  - MANY-2-ONE Mapping.
  - MANY-2-MANY Mapping.
  - Limitations
- Geo Spatial function support
- Capabilities
- Native Queries
  - Direct Query Procedure

Who is the primary audience for this translator?

The above may not satisfy every user’s needs. The document structure in MongoDB can be more complex than what Teiid can currently define. We hope this will eventually catch up in future versions of Teiid. This is currently designed for:

1. Users that are using relational databases and would like to move/migrate their data to MongoDB to take advantages scaling and performance with out modifying the end user applications currently running.
2. Users that are starting out with MongoDB and do not have experience with MongoDB, but are seasoned SQL developers. This provides a low barrier of entry compared to using MongoDB directly as an application developer.
3. Integrate other enterprise data sources with MongoDB based data.

Usage

The name of the translator to use in vdb.xml is "*mongodb*". For example:

```
<vdb name="nothwind" version="1">
    <model name="northwind">
        <source name="local" translator-name="mongodb" connection-jndi-name="java:/mongoDS"/>
    </model>
<vdb>
```

The translator does not provide a connection to the MongoDB. For that purpose, Teiid has a JCA adapter that provides a connection to MongoDB using the MongoDB Java Driver. To define such connector, use the following XML fragment in standalone-teiid.xml. See a example in "<jboss-as>/docs/teiid/datasources/mongodb"

```
    <resource-adapters>
        <resource-adapter id="mongodb">
            <module slot="main" id="org.jboss.teiid.resource-adapter.mongodb"/>
            <transaction-support>NoTransaction</transaction-support>
            <connection-definitions>
                <connection-definition class-name="org.teiid.resource.adapter.mongodb.MongoDBManagedConnectionF
actory"
                        jndi-name="java:/mongoDS"
                        enabled="true"
                        use-java-context="true"
                        pool-name="teiid-mongodb-ds">

                      <!-- MongoDB server list (host:port[;host:port...]) -->
                      <config-property name="RemoteServerList">localhost:27017</config-property>
                      <!-- Database Name in the MongoDB -->
                      <config-property name="Database">test</config-property>
                        <!--
                            Uncomment these properties to supply user name and password
                        <config-property name="Username">user</config-property>
                        <config-property name="Password">user</config-property>
                        -->
                </connection-definition>
            </connection-definitions>
        </resource-adapter>
    </resource-adapters>
```

The above defines the translator and connector. For more ways to create the connector see MongoDB Data Sources. MongoDB translator can derive the metadata based on existing document collections in some scenarios, however when working with complex documents the interpretation of metadata may be inaccurate, in those situations the user MUST define the metadata. For example, you can define a schema using DDL:

```
<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 INSERT operation below executed against table using Teiid,

```
    INSERT INTO Customer(customer_id, FirstName, LastName) VALUES (1, 'John', 'Doe');
```

MongoDB translator will create a below document in the MongoDB

```
{
  _id: ObjectID("509a8fb2f3f4948bd2f983a0"),
  customer_id: 1,
  FirstName: "John",
  LastName: "Doe"
}
```

If a PRIMARY KEY is defined on the table as

```
    CREATE FOREIGN TABLE  Customer (
        customer_id integer PRIMARY KEY,
        FirstName varchar(25),
        LastName varchar(25)
    ) OPTIONS(UPDATABLE 'TRUE');
```

then that column name is automatically used as "_id" field in the MongoDB collection, then document structure is stored in the MongoDB as

```
{
  _id: 1,
  FirstName: "John",
  LastName: "Doe"
}
```

If you defined the composite PRIMARY KEY on Customer table as

```
    CREATE FOREIGN TABLE  Customer (
        customer_id integer,
        FirstName varchar(25),
        LastName varchar(25),
        PRIMARY KEY (FirstName, LastName)
    ) OPTIONS(UPDATABLE 'TRUE');
```

the document structure will be

```
{
  _id: {
         FirstName: "John",
         LastName:  "Doe"
       },
  customer_id: 1,
}
```

Data Types

MongoDB translator supports automatic mapping of Teiid data types into MongoDB data types, including the support for Blobs, Clobs and XML. The LOB support is based on GridFS in MongoDB. Arrays are in the form of

```
{
  _id: 1,
  FirstName: "John",
  LastName: "Doe"
  Score: [89, "ninety", 91.0]
}
```

are supported. User can get individual items in the array using function array_get, or can transform the array into tabular structure using ARRATTABLE.

Note	Note that even though embedded documents can also be in arrays, the handling of embedded documents is different from array with scalar values.

Regular Expressions, MongoDB::Code, MongoDB::MinKey, MongoDB::MaxKey, MongoDB::OID is not currently supported.

Note	Documents that contain values of mixed types for the same key, for example "key" is a string value in one document and an integer in another, the column must be marked as unsearchable as MongoDB will not correct match predicates against the column. See also the importer.sampleSize property.

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
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

Extension Metadata Properties To Build Complex Documents

Using the above DDL or any other metadata facility, a user can map a table in a relational store into a document in MongoDB, however to make effective use of MongoDB, you need to be able to build complex documents, that can co-locate related information, so that data can queried in a single MongoDB query. Otherwise, since MongoDB does not support join relationships like relational database, you need to issue multiple queries to retrieve and join data manually. The power of MongoDB comes from its "embedded" documents and its support of complex data types like arrays and use of the aggregation framework to be able to query them. This translator provides way to achieve that goals.

When you do not define the complex embedded documents in MongoDB, Teiid 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. 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, Teiid 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.

Note	**Usage** - Please note a given table can contain either the "teiid_mongo:EMBEDDABLE" property or the "teiid_mongo:MERGE" property defining the type of nesting in MongoDB. A table is not allowed to have both properties.

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	Note that the Foreign Key defined on child table, must refer to Primary Keys on both 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 have DDL like

```
    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');
```

in the above 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, Teiid 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

- Currently nested embedding of documents has limited support due to capabilities of handling nested arrays is limited in the MongoDB. Nesting of "EMBEDDALBLE" 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 have used either of "EMBEDDABLE" or "MERGE" property, otherwise the query will result in error. In order for Teiid to correctly plan and support the 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, Teiid 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 Teiid engine, without the above property it will result in an error.

When you use above properties and carefully design the MongoDB document structure, Teiid translator can intelligently collate data based on their co-location and take advantage of it while querying.

Geo Spatial function support

MongoDB translator supports geo spatial query operators in the "WHERE" clause, when the data is stored in the GeoJSon format in the MongoDB Document. The supported functions are

```
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 boole
an;
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) RET
URNS 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 above functions 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":"Polygo
n"}'))
```

There are various "st_geom.." methods are available in the Geo Spatial function library in Teiid.

Capabilities

MongoDB translator designed on top of the MongoDB aggregation framework, use of MongoDB version that supports this framework is mandatory. Apart from SELECT queries, this translator also supports INSERT, UPDATE and DELETE queries.

This translator supports

- grouping
- matching
- sorting
- filtering
- limit
- support for LOBs using GridFS
- Composite primary and foreign keys.

Note	**example** - For a full example see https://github.com/teiid/teiid/blob/master/connectors/translator-mongodb/src/test/resources/northwind.ddl

Native Queries

MongoDB source procedures may be created using the teiid_rel:native-query extension - see Parameterizable Native Queries. 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 this feature, override the execution property called _SupportsDirectQueryProcedure to true.

By default the name of the procedure that executes the queries directly is called **native**. Override the execution property _DirectQueryProcedureName to change it to another name.

The MongoDB translator provides a procedure to execute any ad-hoc aggregate query directly against the source without Teiid parsing or resolving. Since the metadata of this procedure's results are not known to Teiid, 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 Teiid 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 }}"
```

OData Translator

The OData translator, known by the type name *"odata"* exposes the OData V2 and V3 data sources and uses the Teiid WS resource adapter for making web service calls. This translator is extension of *Web Services Translator*.

Note	**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 OASIS group, with the help from the OData4J framework, Teiid maps OData entities into relational schema. Teiid supports reading of CSDL (Conceptual Schema Definition Language) from the OData endpoint provided and converts the OData schema into relational schema. The below table shows the mapping selections in OData Translator from CSDL document

OData	Mapped to Relational Entity
EntitySet	Table
FunctionImport	Procedure
AssosiationSet	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.

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 and Function Imports were local to the Teiid system.

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	Supports $count	true
SupportsOdataFilter	Supports $filter	true
SupportsOdataOrderBy	Supports $orderby	true

SupportsOdataSkip	Supports $skip	true
SupportsOdataTop	Supports $top	true

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 only import tables and views from NetflixCatalog.

```
<property name="importer.schemaNamespace" value="System.Data.Objects"/>
<property name="importer.entityContainer" value="NetflixCatalog"/>
```

Note	**OData Server is not fully compatible** - Sometimes it's possible that the odata server you are querying does not fully implement all OData specification features. If your OData implementation does not support a certain feature, then turn off the corresponding capability using "execution Properties", so that Teiid will not pushdown invalid queries to the translator. For example, to turn off $filter you add following to your vdb.xml

```
    <translator name="odata-override" type="odata">
    <property name="SupportsOdataFilter" value="false"/>
    </translator>
```

then use "odata-override" as the translator name on your source model.

Tip	**Native Queries** - Native or direct query execution is not supported through OData translator. However, user can use Web Services Translator's *invokehttp* method directly to issue a Rest based call and parse results using SQLXML.

Tip	**Want to use as Server?** - Teiid can not only consume OData based data sources, but it can expose any data source as an Odata based webservice. For more information see OData Support.

JCA Resource Adapter

The resource adapter for this translator is a Web Service Data Source.

OData V4 Translator

The OData V4 translator, known by the type name *"odata4"* exposes the OData Version 4 data sources and uses the Teiid WS resource adapter for making web service calls. This translator is extension of *Web Services Translator*. Do not use the OData V4 translator against older OData V1-3 sources, instead just use the OData translator.

Note	**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 OASIS group, with the help from the Olingo framework, Teiid maps OData V4 CSDL (Conceptual Schema Definition Language) document from the OData endpoint provided and converts the OData metadata into Teiid's relational schema. The below table shows the mapping selections in OData V4 Translator from CSDL document

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 Teiid system.

It is not recommended to define your own metadata using Teiid DDL for complex services as there are several extension metadata properties required for proper functioning. On non-string properties a NATIVE_TYPE property is expected and should specify the ful 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>
```

The required resource-adapter configuration will look like

```
<resource-adapter id="trippin">
    <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:/tripDS" enabled="true" use-java-context="true" pool-name="teiid-trip-ds">
            <config-property name="EndPoint">
                http://services.odata.org/V4/(S(va3tkzikqbtgu1ist44bbft5))/TripPinServiceRW
            </config-property>
        </connection-definition>
    </connection-definitions>
</resource-adapter>
```

Once you configure above resource-adapter and deploy the VDB successfully, then you can connect to the VDB deployed using Teiid 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 t
hat People_UserName is implicitly added by Teiid metadata)
EXEC GetNearestAirport(lat, lon) ;
```

Configuration of Translator

Execution Properties

Execution properties extend/limit the functionality of the translator based on the physical source capabilities. Sometimes default properties need to adjusted for proper execution of the translator.

Execution Properties

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

Sometimes it’s possible that the odata server you are querying does not fully implement all OData specification features. If your OData implementation does not support a certain feature, then turn off the corresponding capability using "execution Properties", so that Teiid will not pushdown invalid queries to the translator. For example, to turn off $filter you add following to your vdb.xml

```
<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

Importer properties define the behavior of the translator during the metadata import from the physical source.

Importer Properties

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, as if it does not find one configured the translator will select the default name of the EntityContainer.

JCA Resource Adapter

The resource adapter for this translator is a Web Service Data Source.

Note	**Native Queries** - Native or direct query execution is not supported through OData translator. However, user can use Web Services Translator's *invokehttp* method directly to issue a Rest based call and parse results using SQLXML.

Note	**Want to use as OData Server?** - Teiid can not only consume OData based data sources, but it can expose any data source as an OData based webservice. For more information see OData Support.

Swagger Translator

The Swagger translator, known by the type name *"swagger"* exposes the Swagger data sources in realational concepts and uses the Teiid WS resource adapter for making web service calls.

Note	**What is Swagger** - http://swagger.io/ [OpenAPI Specification (Swagger)] Swagger 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 Swagger in almost every modern programming language and deployment environment. With a Swagger-enabled API, you get interactive documentation, client SDK generation and discoverability.

Starting January 1st 2016 the Swagger Specification has been donated to the Open API Initiative (OAI) and has been renamed to the OpenAPI Specification.

Usage

Usage of a Swagger source is similar any other translator in Teiid. The metadata import is supported through the translator, the metadata is imported from source system's swagger.json file and then API from this file is exposed as stored procedures in Teiid, then source system can be queried by executing these stored procedures in Teiid system.

Note	Parameter order is guaranteed by the swagger libraries. It is recommended that you call procedures using named, rather than positional parameters, if you rely upon the native import.

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">
        <source name="s" translator-name="swagger" connection-jndi-name="java:/swagger"/>
    </model>
</vdb>
```

The required resource-adapter configuration will look like

```
<resource-adapter id="swagger">
    <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:/swagger" enabled="true" use-java-context="true" pool-name="teiid-swagger-ds">
            <config-property name="EndPoint">
                http://petstore.swagger.io/v2
            </config-property>
        </connection-definition>
    </connection-definitions>
</resource-adapter>
```

Once you configure above resource-adapter and deploy the VDB successfully, then you can connect to the VDB deployed using Teiid JDBC driver and issue SQL statements like

```
EXEC findPetsByStatus(('sold',))
EXEC getPetById(1461159803)
EXEC deletePet('', 1461159803)
```

Configuration of Translator

Execution Properties

Execution properties extend/limit the functionality of the translator based on the physical source capabilities. Sometimes default properties need to adjusted for proper execution of the translator.

Execution Properties

none

Importer Properties

Importer properties define the behavior of the translator during the metadata import from the physical source.

Importer Properties

Name	Description	Default
useDefaultHost	Use default host specified in the Swagger file; Defaults to true, when false uses the endpoint in the resource-adapter	true
preferredScheme	Preferred Scheme to use when Swagger file supports multiple invocation schemes like http, https	null
preferredProduces	Preferred Accept MIME type header, this should be one of the Swagger 'produces' types;	application/json
preferredConsumes	Preferred Content-Type MIME type header, this should be one of the Swagger 'consumer' types;	application/json

Example importer settings to avoid calling host defined on the swagger.json file

```
<property name="importer.useDefaultHost" value="false"/>
```

JCA Resource Adapter

The resource adapter for this translator is a Web Service Data Source.

Note	**Native Queries** - Native or direct query execution is not supported through Swagger translator. However, user can use Web Services Translator's *invokehttp* method directly to issue a Rest based call and parse results using SQLXML.

Limitations

- "application/xml" mime type in both "Accept" and "Content-Type" is currently not supported
- File, Map proeprties are currently not supported, thus any multi-part payloads are not supported
- Security metadata is currently not supported
- Custom properties that start with "x-" are not supported.
- Schema with "allof", "multipleof", "items" from JSON schema are not supported

OLAP Translator

The OLAP Services translator, known by the type name *olap*, exposes stored procedures for calling analysis sevices backed by a OLAP server using MDX query language. This translator exposes a stored procedure, invokeMDX, that returns a result set containing tuple array values for a given MDX query. invokeMDX will commonly be used with the ARRAYTABLE table function to extract the results.

Since the Cube metadata exposed by the OLAP servers and relational database metadata are so different, there is no single way to map the metadata from one to other. It is best to query OLAP system using its own native MDX language through. MDX queries my be defined statically or built dynamically in Teiid's abstraction layers.

Usage

The olap translator exposes one low level procedure for accessing olap services.

InvokeMDX Procedure

`invokeMdx` returns a resultset of the tuples as array values.

```
Procedure invokeMdx(mdx in STRING, params VARIADIC OBJECT) returns table (tuple object)
```

The mdx parameter is a MDX query to be executed on the OLAP server.

The results of the query will be returned such that each row on the row axis will be packed into an array value that will first contain each hierarchy member name on the row axis then each measure value from the column axis.

The use of Data Roles should be considered to prevent arbitrary MDX from being submitted to the invokeMDX procedure.

Native Queries

OLAP source procedures may be created using the teiid_rel:native-query extension - see Parameterizable Native Queries.

The parameter value substitution directly inserts boolean, and number values, and treats all other values as string literals.

The procedure will invoke the native-query similar to an invokeMdx 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

The invokeMdx procedure is the direct query procedure for the OLAP translator. It may be disabled or have it's name changed via the common direct query translator properties just like any other source. A call to the direct query procedure without any parameters will not attempt to parse the mdx query for parameterization. If parameters are used, the value substitution directly inserts boolean, and number values, and treats all other values as string literals.

JCA Resource Adapter

The resource adapter for this translator provided through data source in WildFly, Refer to Admin Guide for "JDBC Data Sources" configuration section. Two sample xml files are provided for accessing OLAP servers in the teiid-examples section. One is Mondrian specific, when Mondrian server is deployed in the same WildFly as Teiid (mondrian-ds.xml). To access any other OLAP servers using XMLA interface, the data source for them can be created using them example template olap-xmla-ds.xml

Note	Due to a classloading change with Mondrian 3.6 and later, a workaround is needed to use a later driver - TEIID-4617 The olap translator module.xml under modules/system/layers/dv/org/jboss/teiid/translator/olap/main/ needs to have a dependency to the Mondrian driver module.

Salesforce Translators

The Salesforce translator supports the SELECT, DELETE, INSERT, UPSERT, and UPDATE operations against a Salesforce.com account. It is designed for use with the Teiid Salesforce resource adapter.

Salesforce API Version Support

salesforce

The translator, known by the type name ***salesforce***, provides Salesforce API 34.0 support. The translator must be used with the corresponding Salesforce resource adapter of the same API version. Salesforce API version 34.0 support has been removed.

salesforce-34

The translator, known by the type name of ***salesforce-34***, provides Salesforce API 34.0 support. The translator must be used with the corresponding Salesforce resource adapter of the same API version.

salesforce-41

The translator, known by the type name of ***salesforce-41***, provides Salesforce API 41.0 support. The translator must be used with the corresponding Salesforce resource adapter of the same API version.

Other API Versions

If you need connectivity to an API version other than what is built in, please utilize the project https://github.com/teiid/salesforce to generate new resource adapter / translator pair.

Execution Properties

Name	Description	Default
ModelAuditFields	Add Audit Fields To Model (the import property takes precedence)	false
MaxBulkInsertBatchSize	Batch Size to use to insert bulk inserts	2048
SupportsGroupBy	Supports Group By Pushdown. Set to false to have Teiid process group by aggregations, such as those returning more than 2000 rows which error in SOQL.	true

The Salesforce translator can import metadata.

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
ModelAuditFields	Add Audit Fields To Model	false	n/a

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. Please raise an issue if you cannot successfully access a specific API version.

Extension Metadata Properties

Salesforce is not relational database, however Teiid provides ways to map Saleforce data into relational constructs like Tables and Procedures. You can define a foreign Table using DDL in Teiid VDB, which maps to Salesforce's SObject. At runtime, to interpret this table back to a SObject, Teiid decorates or tags this table definition with additional metadata. For example, a table is defined as

```
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 above the property in OPTIONS clause with property "teiid_sf:Supports Query" annotating that this tables supports SELECT commands. The below are list of metadata extension properties that can be used on Salesforce schema.

Property Name	Description	Required	Default	Applies To
Supports Query	The table supports SELECT commands	false	true	Table
Supports Retrieve	The table supports retrieval of results as result of SELECT commands	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 Teiid Query Planner, the Salesforce connector supports nearly all of the SQL syntax supported by the Teiid.

The Salesforce Connector executes SQL commands by "pushing down" the command to Salesforce whenever possible, based on the supported capabilities. Teiid will automatically provide additional database functionality when the Salesforce Connector does not explicitly provide support for a given SQL construct. In cases where certain SQL capabilities cannot be pushed down to Salesforce, Teiid will push down the capabilities that are supported, and fetch a set of data from Salesforce. Then, Teiid will evaluate the additional capabilities, creating a subset of the original data set. Finally, Teiid will pass the result to the client.

If you are issuing queries with a group by clause and receive an error for salesforce related to queryMore not being supported, you may 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, but they do support CompareCriteriaEquals, so the query that is passed to Salesforce by the connector will be transformed to this query.

```
SELECT Reports FROM Supervisor where Division = 'customer support';
```

The array_agg() function will be applied by the Teiid Query Engine to the result set returned by the connector.

In some cases multiple calls to the Salesforce application will be made to support the SQL passed to the connector.

```
DELETE From Case WHERE Status = 'Closed';
```

The API in Salesforce to delete objects only supports deleting by ID. 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 unsupported capabilities, in order to avoid fetching large data sets from Salesforce and making you queries as performant as possible. See all Supported 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. The full Salesforce documentation for selecting from mullti-select picklists can be found at the following link Querying Mulit-select Picklists

Teiid SQL does not support the includes or excludes operators, but the Salesforce connector provides user defined function definitions for these operators that provided 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

The Salesforce connector supports the calling 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 behaviour 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' the connector will call queryAll.

```
select * from Contact where isDeleted = true;
```

If the isDeleted column is used as a parameter in the query, and the value is 'false' the connector perform the default behavior will call 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

Salesforce does not support joins like a relational database, but it does have support for queries that include parent-to-child or child-to-parent relationships between objects. These are termed Relationship Queries. The SalesForce connector supports Relationship Queries 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

SalesForce translator also supports bulk insert statements 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.xml 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 quering large tables (typically over 10,000,000 records) or if experiencing timeouts with just result batching, Teiid can issue queries to Salesforce using the bulk API. When using a bulk select, PK chunking will be enabled if supported by 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 by Salsforce API equal to greater than 28.

Supported Capabilities

The following are the capabilities supported by 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 - Supported for String fields only.
- RowLimit
- Basic Aggregates
- OuterJoins with join criteria KEY

Native Queries

Salesforce procedures may optionally have native queries associated with them - see Parameterizable Native Queries. 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 SF 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 this feature, override the execution property called _SupportsDirectQueryProcedure to true.

Tip	By default the name of the procedure that executes the queries directly is **native**. Override the execution property _DirectQueryProcedureName to change it to another name.

The Salesforce translator provides a procedure to execute any ad-hoc SOQL query directly against the source without Teiid parsing or resolving. Since the metadata of this procedure's results are not known to Teiid, they are returned as an object array. ARRAYTABLE can be used construct tabular output for consumption by client applications. Teiid exposes this procedure with a simple query structure as follows:

Select

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 - see Parameterizable Native Queries

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

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 or Update

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 user can + set override execution property vdb.xml file to change it.

JCA Resource Adapter

The resource adapter for this translator is provided through Salesforce Data Sources. Refer to Admin Guide for configuration.

SAP Gateway Translator

The SAP Gateway Translator, known by the type name *sap-gateway*, provides a translator for accessing the SAP Gateway using the OData protocol. This translator is extension of OData Translator and uses Teiid WS resource adapter for making web service calls. This translator understands the most of the SAP specific OData extensions to the metadata defined in the document SAP Annotations for OData Version 2.0

When the metadata is imported from SAP Gateway, the Teiid models are created to accordingly for SAP specific *EntitySet* and *Property* annotations defined in document above.

The following "execution properties" are supported in this translator

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	Supports $count	true
SupportsOdataFilter	Supports $filter	true
SupportsOdataOrderBy	Supports $orderby	true
SupportsOdataSkip	Supports $skip	true
SupportsOdataTop	Supports $top	true

Based on how you implemented your SAP Gateway service, if can choose to turn off some of the features above.

Note	**Using pagable, topable metadata extensions?** - If metadata on your service defined "pagable" and/or "topable" as "false" on **any** table, you **must** turn off "SupportsOdataTop" and "SupportsOdataSkip" execution-properties in your translator, so that you will not end up with wrong results. SAP metadata has capability to control these in a fine grained fashion any on EnitySet, however Teiid can only control these at translator level.

Note	**SAP Examples** - Sample examples defined at http://scn.sap.com/docs/DOC-31221, we found to be lacking in full metadata in certain examples. For example, "filterable" clause never defined on some properties, but if you send a request $filter it will silently ignore it. You can verify this behavior by directly executing the REST service using a web browser with respective query. So, Make sure you have implemented your service correctly, or you can turn off certain features in this translator by using "execution properties" override. See an example in OData Translator

Web Services Translator

The Web Services translator, known by the type name *ws*, exposes stored procedures for calling web services backed by a Teiid WS resource adapter. The WS resource adapter may optionally be configured to point at a specific WSDL. 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 ws 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 WS translator exposes low level procedures for accessing web services. See also the twitter example in the kit.

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) re
turns 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 WS resource adapter 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 ou
t 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 WS resource adapter 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 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 may be necessary if 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.

Note	**Native queries** - Native queries or a direct query execution procedure is not supported on 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 may perform validation which results in the stream being consumed.

JCA Resource Adapter

The resource adapter for this translator is a Web Service Data Source.

Note	**WS-Security** - Currently you can only use WSDL based Procedures participate in WS-Security, when resource-adapter is configured with correct CXF configuration.

Federated Planning

Teiid at its core 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 in a single SQL query. This allows you to focus on building your application, not on hand-coding joins, and other relational operations, between data sources.

Child Pages

- Planning Overview
- Query Planner
- Query Plans
- Federated Optimizations
- Subquery Optimization
- XQuery Optimization
- Federated Failure Modes
- Conformed Tables

Planning Overview

When the query engine receives an incoming SQL query it performs the following operations:

1. **Parsing** - validate syntax and convert to internal form
2. **Resolving** - link all identifiers to metadata and functions to the function library
3. **Validating** - validate SQL semantics based on metadata references and type signatures
4. **Rewriting** - rewrite SQL to simplify expressions and criteria
5. **Logical plan optimization** - the rewritten canonical SQL is converted into a logical plan for in-depth optimization. The Teiid 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, Teiid also takes advantage of costing information. The logical plan optimization steps can be seen by using SET SHOWPLAN DEBUG clause, a sample steps are described in Query Planner#Reading a Debug Plan. More details about logical plan's nodes and rule-based optimization refer to Query Planner.
6. **Processing plan conversion** - the logic plan is converted into an executable form where the nodes are representative of basic processing operations. The final processing plan is displayed as the Query Plans.

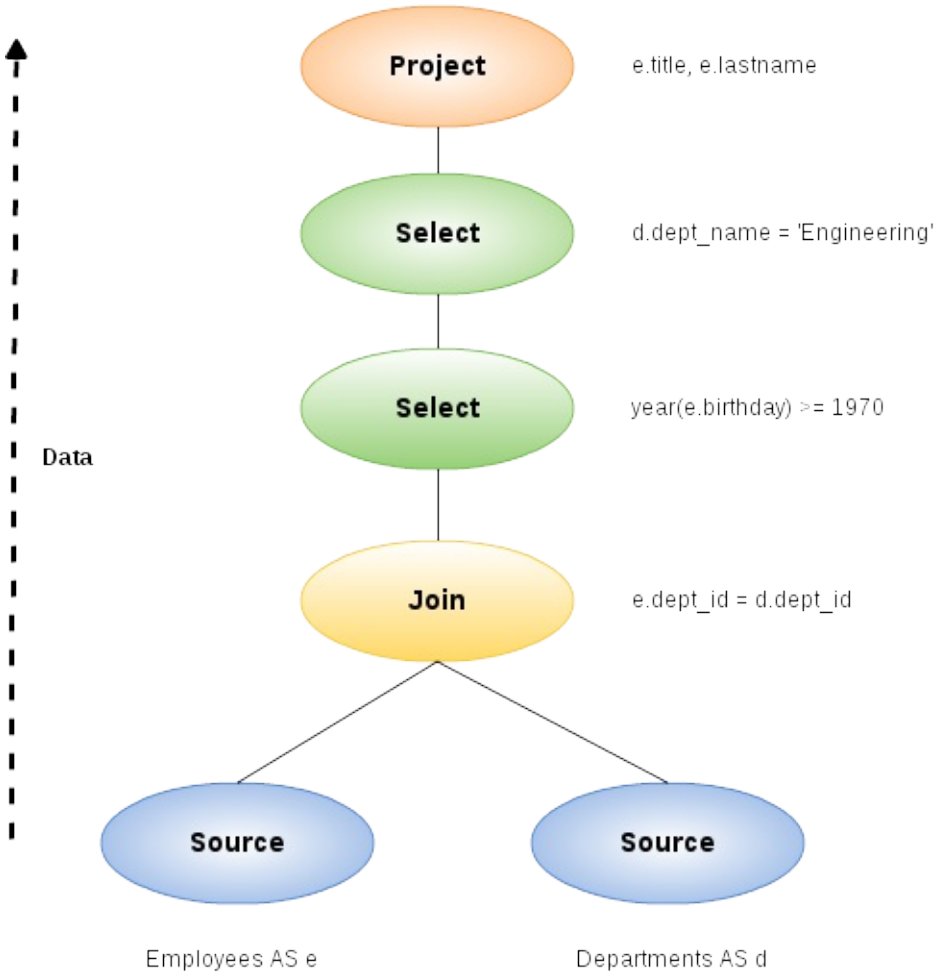
The logical query plan is a tree of operations used to transform data in source tables to the expected result set. In the tree, data flows from the bottom (tables) to the top (output). The primary logical operations are *select* (select or filter rows based on a criteria), *project* (project or compute 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

```
SELECT e.title, e.lastname FROM Employees AS e JOIN Departments AS d ON e.dept_id = d.dept_id WHERE year(e.birt
hday) >= 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**, 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 does 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.

Query Planner

- Canonical Plan and All Nodes
- Node Properties
  - Access Properties
  - Set operation Properties
  - Join Properties
  - Project Properties
  - Select Properties
  - Sort Properties
  - Source Properties
  - Group Properties
  - Tuple Limit Properties
  - General and Costing Properties
- Rules

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.

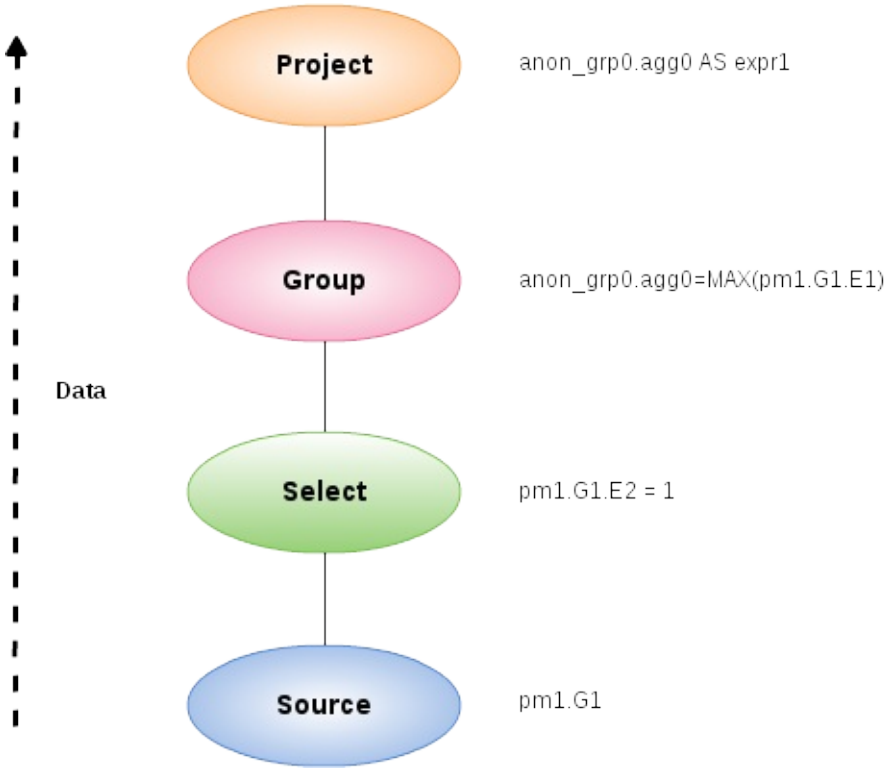
Canonical Plan and All Nodes

As Planning Overview, a user SQL statement after Parsing, Resolving, Validating, Rewriting, it be 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) - handled by a specialized PROJECT NODE
2. FROM (read and join all data from tables) - SOURCE node for each from clause item, Join node (if >1 table)
3. WHERE (filter rows) - SELECT node
4. GROUP BY (group rows into collapsed rows) - GROUP node

5. HAVING (filter grouped rows) - SELECT node
6. SELECT (evaluate expressions and return only requested rows) - PROJECT node and DUP_REMOVE node (for SELECT DISTINCT)
7. INTO - specialized PROJECT with a SOURCE child
8. ORDER BY (sort rows) - SORT node
9. LIMIT (limit result set to a certain range of results) - 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 affect of grouping generates what is effectively an inline view, anon_grp0, to handle the projection of values created by the grouping.

Table 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, etc.)

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, etc.
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.

Access Properties

Table 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

Set operation Properties

Table 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.

Join Properties

Table 4. **Join Properties**

Property Name	Description
JOIN_CRITERIA	all join predicates
JOIN_TYPE	type of join (INNER, LEFT OUTER, etc.)
JOIN_STRATEGY	the algorithm to use (nested loop, merge, etc.)
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
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

Project Properties

Table 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

Select Properties

Table 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
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

Sort Properties

Table 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

Source Properties

Table 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, etc.) defining the source
CORRELATED_REFERENCES	the correlated references for the nodes below the source
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.

Group Properties

Table 9. **Group Properties**

Property Name	Description
GROUP_COLS	the grouping columns
ROLLUP	if the grouping includes a rollup

Tuple Limit Properties

Table 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

General and Costing Properties

Table 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, etc.
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 max 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, i.e. 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 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 Federated Optimizations#Partitioned Union. 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 Federated Optimizations#Access Patterns.
- Plan Joins - this rule attempts to find an optimal ordering of the joins performed in the plan, while ensuring that Federated Optimizations#Access Patterns dependencies are met. This rule has three main steps. First it must determine an ordering of joins that satisfy the access patterns present. Second 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). Third 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 Teiid 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 directly supported by the translator 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)
```

This plan tree must have the criteria remain above the join, since 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 possible. 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, etc.) and is also influenced by the presence of constraints (unique, primary key, index, etc.). 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, OUTP
UT_COLS=[e1]})
    Source(groups=[pm1.g1])
```

Some important planning decisions are shown in the plan as they occur as an annotation. For example the snippet below shows that the access node could not be raised as 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=nu
ll})
    Access(groups=[pm1.g1], props={SOURCE_HINT=null, MODEL_ID=Schema name=pm1, nameInSource=null, uuid=3341, OU
TPUT_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.g
1) 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=nu
ll})
    Access(groups=[pm1.g1], props={SOURCE_HINT=null, MODEL_ID=Schema name=pm1, nameInSource=null, uuid=3341, OU
TPUT_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. Document projection is the most common optimization. It will be shown in the debug plan as an annotation. For example with the user query containing "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()
```

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 SQL options available are as follows:

SET SHOWPLAN [ON|DEBUG]- Returns the processing plan or the plan and the full planner Debug Log. See also the SET Statement.

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.

Retrieving a Query Plan Using Teiid Extensions

```
statement.execute("set showplan on");
ResultSet rs = statement.executeQuery("select ...");
TeiidStatement 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"));
```

The query plan is made available automatically in several of Teiid's tools.

Analyzing a Query Plan

Once a query plan has been obtained you will most commonly be looking for:

- 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, etc.
- Presence of federated optimizations, such as dependent joins.

- Ensure hints have the desired affects - see Hints and Options, hints in the FROM Clause, Subquery Optimization, and Federated Optimizations.

All of the above information can be determined 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 - see the XQuery section below or 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 same plan in xml form looks like:

```
<?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. From the Debug Log the same plan as above would appear as:

```
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=[pm
1.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
```

Node Properties

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 - the 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, etc.)
- 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, etc.)
- 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.

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 Teiid though the Connector API. Any work not performed at the source is then processed in Federate's relational engine.

Based upon capabilities, Teiid will manipulate the query plan to ensure that each source performs as much joining, filtering, grouping, etc. as possible. In may cases, such as with join ordering, planning is a combination of Standard Relational Techniques and, cost based 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. See Query Plans on how to read a plan to ensure that source queries are as efficient as possible.

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. There are three different kinds of dependent joins that Teiid supports:

- Join based on in/equality support - where the engine will determine how to break up large queries based upon 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 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.

Teiid supports hints 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. MAKEDEP as a non-comment hint supports optional max and join arguments - MAKEDEP(JOIN) meaning that the entire join should be pushed, and MAKEDEP(MAX:5000) meaning that the dependent join should only be performed if there are less than the max number of values from the independent side.

- MAKENOTDEP - prevents the clause from being the dependent side of a join.

Theses 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/MAKEIND hint 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/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. If you are working with custom translators see Dependent Join Pushdown as to how to support it key and full pushdown.

Note	**Key/Full Pushdown** is currently only supported out-of-the box by a subset of JDBC translators. To enable support, set the translator override property "enableDependentJoins" to "true". The JDBC source must support the creation of temporary tables, which typically requires a Hibernate dialect. Translators that should support this feature include: 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

Teiid 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, e.g. "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

Teiid user and transformation queries can contain a meta source hint that can provide additional information to source queries. The source hint has the 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 may 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. See the Developer's Guide for more on using an ExecutionContext.
- 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. See also a blog on the join optimization.

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 property implicit_partition.columnName to name of the partitioning column used on each partitioned view in the model/schema.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<vdb name="partition" version="1">
    <model name="all_customers" type="VIRTUAL">
        <property name="implicit_partition.columnName" value="theColumn"/>
    ...
```

Standard Relational Techniques

Teiid 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 Teiid will attempt to form an appropriate left outer join. These sources are restricted to only supporting 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.

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 Teiid to perform row-by-row compensating processing.
- The MJ hint directs the optimizer to use a traditional, semijoin, or antisemijoin merge join if possible. The 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 (merge join), DJ (dependent join), 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 (merge join) or NO_UNNEST (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.

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, e.g. //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', chil
d_val integer)
```

The context XQuery and the column path expression allow the streaming optimization, rather than loading the entire doc 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.

Partial Results

Teiid 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 associated with the source issues 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. See the Client Guide for more on Partial Results Mode and SQLWarnings.

Conformed Tables

A conformed table is a source table that is the same in several physical sources. Unlike Multisource Models which assume a partitioning paradigm, the planner assumes any conformed table may be substituted for another to improve performance. 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

```
{http://www.teiid.org/ext/relational/2012}conformed-sources
```

extension metadata property to the appropriate source tables. Extension properties can be set in the vdb.xml when using full DDL Metadata or alter statements, or at runtime using the setProperty system procedure. The property is expected to be a comma separated list of physical model/schema names.

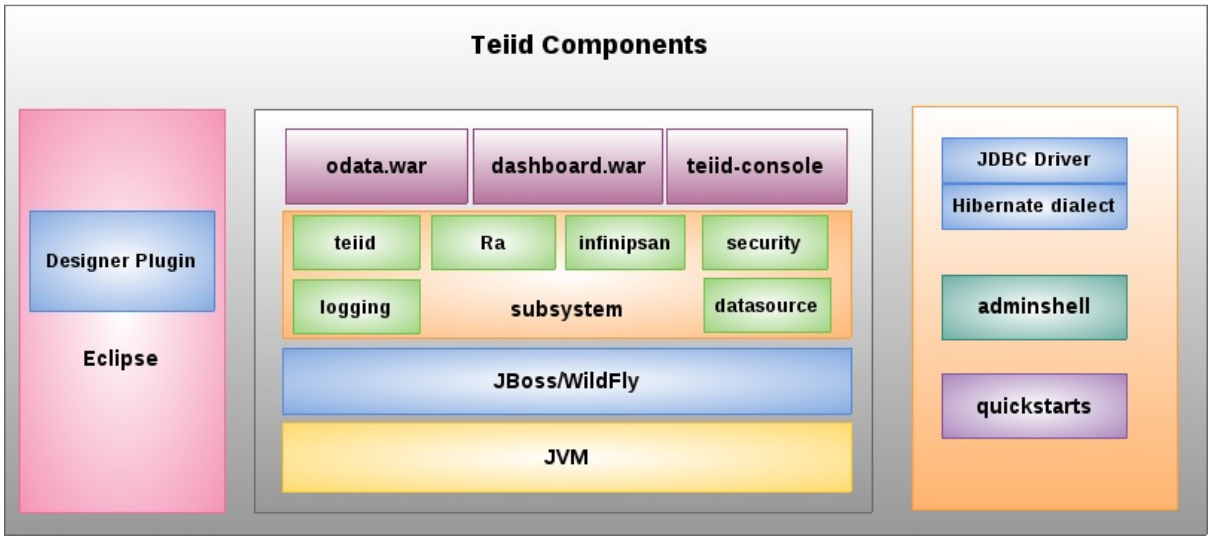
DDL Alter Example

```
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. Just as with the multi-source feature, there is then no source specific metadata entry to the conformed sources. Also just as with multi-source planning, the capabilities are assumed to be the same across conformed sources.

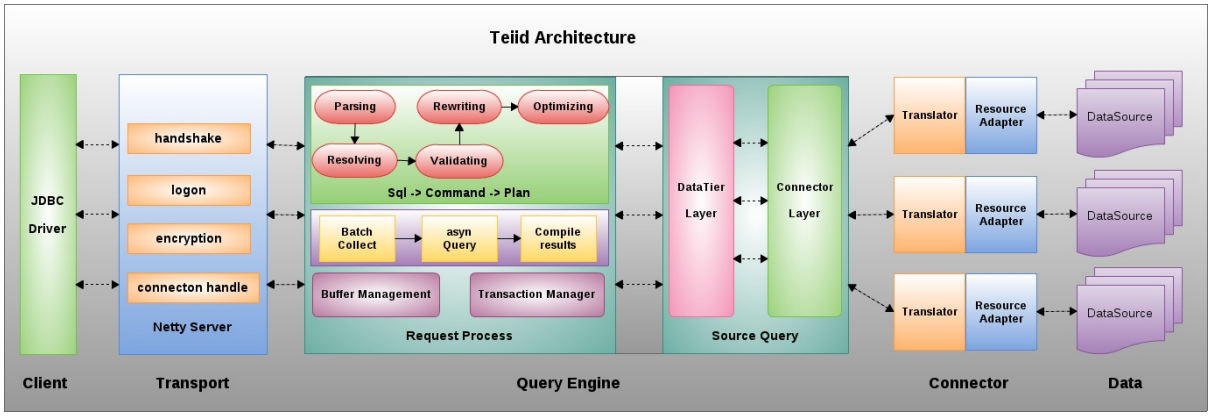
The engine will take the list of conformed sources and associate a set of model metadata ids to the corresponding access node. The logic considering joins and subqueries will also consider the conformed sets when making pushdown decisions. The subquery handling will only check for conformed sources for the subquery - not in the parent. So having a conformed table in the subquery will pushdown as expected, but not vice versa.

Teiid Components



- **Designer Plugin** - Deprecated Eclipse Plugin based Teiid design environment, used to connect/federate/transform datasources to produce a `.vdb` file.
- **JVM** - Teiid is a pure Java Data Virtualization Platform.
- **WildFly** - Teiid use a plugable installation which need a WildFly Server installed, alternatively, a full installed WildFly kit be distributed.
- **Subsystem** - Due to WildFly's Modular and Pluggable Architecture(a series of Management commands compose of a subsystem, a series of subsystems compose of the whole server), Teiid implement WildFly's Controller/Management API developed a `teiid` subsystem and reuse lots of other subsystems like `resource-adapter`, `infinispan`, `security`, `logging`, `datasource`.
- **odata.war** - Teiid support OData via odata.war, more details refer to OData Support
- **dashboard.war** - A web based dashboard generator.
- **teiid-console** - A web based administrative and monitoring tool for Teiid, more details refer to Teiid Console
- **JDBC Driver** - JDBC Driver to connect to Teiid Server.
- **AdminAPI** - An API for performing management and monitoring:../dev/AdminAPI.adoc[AdminAPI]
- **quickstarts** - A maven quickstart showing how to utilize Teiid.

Teiid Architecture



- **Client** - Client Develop Guide
- **Transport** - A Transport managements client connections - security authentication, encryption, etc.
- **Query Engine** - The Query Engine has several layers / components. Request processing at a high level:

    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. Lastly rule and cost-based optimization convert the logical plan to a final Processor Plan. More details refer to 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 Teiid 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/Resource Adapter pair to interact directly with a source. Connectivity is provided for heterogeneous data stores, like Databases/Data warehouse, NoSQL, Hadoop, Data Grid/Cache, File, SaaS, etc. - see Data Sources.

- **Translator** - Teiid has developed a series of Translators, for more details refer to Translators.
- **Resource Adapter** - Provides container managed access to a source, for more details refer to Developing JEE Connectors.

Terminology

- VM or Process – a JBossAS instance running Teiid.
- Host – a machine that is "hosting" one or more VMs.
- Service – a subsystem running in a VM (often in many VMs) and providing a related set of functionality. In addition to these main components, the service platform provides a core set of services available to applications built on top of the service platform. These services are:
- Session – the Session service manages active session information.
- Buffer Manager – the Buffer Manager service provides access to data management for intermediate results.
- Transaction – the Transaction service manages global, local, and request scoped transactions. See also the documentation on Transaction Support.

Data Management

Cursoring and Batching

Teiid cursors all results, regardless of whether they are from one source or many sources, and regardless of what type of processing (joins, unions, etc.) have been performed on the results.

Teiid 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 has a cap on 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 max 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.

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 is dependent on the connector implementation. For example, JDBC connectors will asynchronously call cancel on the underlying JDBC driver, which may or may not actually support this method.

User Query Timeouts

User query timeouts in Teiid 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 may also set a default statement timeout via the connection property "QUERYTIMEOUT". ODBC clients may also utilize QUERYTIMEOUT as an execution property via a set statement to control the default timeout setting. See the Client Developers Guide for more on connection/execution properties and set statements.

Server-side timeouts start when the query is received by the engine. There may be a skew from the when the client issued the query due to network latency or server load that may slow the processing of IO work. The timeout will be cancelled if the first result is sent back before the timeout has ended. See the VDBs section for more on setting the query-timeout VDB property. See the Admin Guide for more on modifying the file to set default query timeout for all queries.

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.

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 are also eligible to be made into dependent joins.

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.

It consists of two phases. The first phase ("sort") will take an unsorted input stream and 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. Since the stream may be more than can fit in memory, this may result in many sorted streams.

The second phase ("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 pass, all input streams are dropped. In this way, each pass reduces the number of sorted streams. When only one stream remains, it is the final output.

BNF for SQL Grammar

- Main Entry Points
  - callable statement
  - ddl statement
  - procedure body definition
  - directly executable statement
- Reserved Keywords
- Non-Reserved Keywords
- Reserved Keywords For Future Use
- Tokens
- Production Cross-Reference
- Productions

Reserved Keywords

Keyword	Usage
ADD	add set child option, add set option, ADD column, ADD constraint
ALL	standard aggregate function, function, Create 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
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, 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
CLOB	simple data type, xml serialize
COLUMN	ADD column, DROP column, ALTER TABLE, Create GRANT, Revoke GRANT
COMMIT	create temporary table
CONSTRAINT	Create 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 role, create schema, create server, aka data source, create table, create temporary table, create trigger, procedure body definition

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 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
DROP	DROP column, drop option, Drop data wrapper, drop option, drop procedure, drop role, drop schema, drop server, aka data source, 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	non numeric literal

FETCH	fetch clause
FILTER	filter clause
FLOAT	simple data type
FOR	for each row trigger action, function, 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, aka data source, Drop data wrapper, drop procedure, drop schema, drop table, foreign key, Import foreign schema
FROM	delete statement, from clause, function, Import foreign schema, is distinct, Revoke GRANT
FULL	qualified table
FUNCTION	create procedure, drop procedure, Create GRANT, Revoke GRANT
GLOBAL	create foreign or global temporary table, drop table
GRANT	Create GRANT
GROUP	function, group by clause
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 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	Create GRANT, object table, Revoke GRANT
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 aggreate 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, is distinct, is null predicate, match predicate, boolean factor, procedure parameter, procedure result column, like regex predicate, in predicate, temporary table element
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
OBJECT	simple data type
OF	alter, ALTER TRIGGER, create trigger
OFFSET	limit clause
ON	alter, ALTER TRIGGER, create foreign temp table, create temporary table, create trigger, Create 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	Create 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, drop procedure, Create GRANT, procedure body definition, Revoke GRANT
RANGE	window frame
REAL	simple data type
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	fetch clause, for each row trigger action, limit clause, text table, window frame bound
ROWS	create temporary table, fetch clause, limit clause, window frame
SECOND	function
SELECT	grant type, select clause
SERVER	ALTER SERVER, create schema, create server, aka data source, drop server, aka data source, Import foreign schema
SET	add set child option, add set option, option namespace, update statement, use 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
TABLE	ALTER TABLE, create procedure, create foreign or global temporary table, create foreign temp table, create temporary table, drop table, drop table, Create GRANT, query primary, Revoke GRANT, table subquery
TEMPORARY	create foreign or global temporary table, create foreign temp table, create temporary table, drop table, Create 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 GRANT, match predicate

TRAILING	function
TRANSLATE	function
TRIGGER	alter, ALTER TRIGGER, create trigger
TRUE	non numeric literal
UNION	cross join, query expression body
UNIQUE	other constraints, inline constraint
UNKNOWN	non numeric literal
UPDATE	alter, ALTER TRIGGER, create trigger, dynamic data statement, grant type, update statement
USER	function
USING	dynamic data statement
VALUES	query primary
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, procedure body definition
WHEN	case expression, searched case expression
WHERE	filter clause, where clause
WHILE	while statement
WITH	assignment statement, create role, Import another Database, query expression, with role, data statement
WITHIN	function
WITHOUT	assignment statement, data statement
WRAPPER	ALTER DATA WRAPPER, create data wrapper, create server, aka data source, Drop data wrapper
XML	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
XMLPI	function
XMLQUERY	xml query
XMLSERIALIZE	xml serialize
XMLTABLE	xml table
XMLTEXT	function
YEAR	function

Non-Reserved Keywords

Name	Usage
ACCESS	Import another Database, non-reserved identifier
ACCESSPATTERN	other constraints, non-reserved identifier
AFTER	alter, create trigger, non-reserved identifier
ARRAYTABLE	array table, non-reserved identifier
AUTO_INCREMENT	alter column options, table element, view element, non-reserved identifier
AVG	standard aggregate function, non-reserved identifier
CHAIN	sql exception, non-reserved identifier
COLUMNS	array table, non-reserved identifier, object table, text table, xml table
CONDITION	Create GRANT, non-reserved identifier, Revoke GRANT

CONTENT	non-reserved identifier, xml parse, xml serialize
CONTROL	Import another Database, non-reserved identifier
COUNT	standard aggregate function, non-reserved identifier
COUNT_BIG	standard aggregate function, non-reserved identifier
CURRENT	non-reserved identifier, window frame bound
DATA	ALTER DATA WRAPPER, create data wrapper, create server, aka data source, Drop data wrapper, non-reserved identifier
DATABASE	ALTER DATABASE, create database, Import another Database, non-reserved identifier, use database
DEFAULT	xml namespace element, non-reserved identifier, object table column, post create column, procedure parameter, xml table column
DELIMITER	non-reserved identifier, text aggreate function, text table
DENSE_RANK	analytic aggregate function, non-reserved identifier
DISABLED	alter, ALTER TRIGGER, non-reserved identifier
DOCUMENT	non-reserved identifier, xml parse, xml serialize
DOMAIN	create a domain or type alias, non-reserved identifier
EMPTY	non-reserved identifier, xml query
ENABLED	alter, ALTER TRIGGER, non-reserved identifier
ENCODING	non-reserved identifier, text aggreate function, xml serialize
EVERY	standard aggregate function, non-reserved identifier
EXCEPTION	compound statement, declare statement, non-reserved identifier
EXCLUDING	non-reserved identifier, xml serialize
EXTRACT	function, non-reserved identifier
FIRST	fetch clause, non-reserved identifier, sort specification
FOLLOWING	non-reserved identifier, window frame bound
GEOGRAPHY	non-reserved identifier, simple data type

GEOMETRY	non-reserved identifier, simple data type
HEADER	non-reserved identifier, text aggreate function, text table column, text table
INCLUDING	non-reserved identifier, xml serialize
INDEX	other constraints, inline constraint, non-reserved identifier
INSTEAD	alter, ALTER TRIGGER, create trigger, non-reserved identifier
JAAS	non-reserved identifier, with role
JSON	non-reserved identifier, simple data type
JSONARRAY_AGG	non-reserved identifier, ordered aggregate function
JSONOBJECT	json object, non-reserved identifier
KEY	create temporary table, foreign key, inline constraint, non-reserved identifier, primary key
LAST	non-reserved identifier, sort specification
LISTAGG	function, non-reserved identifier
MASK	Create GRANT, non-reserved identifier, Revoke GRANT
MAX	standard aggregate function, make dep options, non-reserved identifier
MIN	standard aggregate function, non-reserved identifier
NAME	function, non-reserved identifier, xml element
NAMESPACE	option namespace, non-reserved identifier
NEXT	fetch clause, non-reserved identifier
NONE	non-reserved identifier
NULLS	non-reserved identifier, sort specification
OBJECTTABLE	non-reserved identifier, object table
ORDINALITY	non-reserved identifier, text table column, xml table column
PASSING	non-reserved identifier, object table, xml query, xml query, xml table
PATH	non-reserved identifier, xml table column

POSITION	function, non-reserved identifier
PRECEDING	non-reserved identifier, window frame bound
PRESERVE	create temporary table, non-reserved identifier
PRIVILEGES	Create GRANT, non-reserved identifier, Revoke GRANT
QUERYSTRING	non-reserved identifier, querystring function
QUOTE	non-reserved identifier, text aggreate function, text table
RAISE	non-reserved identifier, raise statement
RANK	analytic aggregate function, non-reserved identifier
RENAME	ALTER PROCEDURE, ALTER TABLE, non-reserved identifier
REPOSITORY	Import foreign schema, non-reserved identifier
RESULT	non-reserved identifier, procedure parameter
ROLE	create role, drop role, non-reserved identifier, with role
ROW_NUMBER	analytic aggregate function, non-reserved identifier
SCHEMA	create schema, drop schema, Create GRANT, Import foreign schema, non-reserved identifier, Revoke GRANT, use schema
SELECTOR	non-reserved identifier, text table column, text table
SERIAL	alter column options, table element, view element, non-reserved identifier, temporary table element
SKIP	non-reserved identifier, text table
SQL_TSI_DAY	time interval, non-reserved identifier
SQL_TSI_FRAC_SECOND	time interval, non-reserved identifier
SQL_TSI_HOUR	time interval, non-reserved identifier
SQL_TSI_MINUTE	time interval, non-reserved identifier
SQL_TSI_MONTH	time interval, non-reserved identifier
SQL_TSI_QUARTER	time interval, non-reserved identifier
SQL_TSI_SECOND	time interval, non-reserved identifier
SQL_TSI_WEEK	time interval, non-reserved identifier

SQL_TSI_YEAR	time interval, non-reserved identifier
STDDEV_POP	standard aggregate function, non-reserved identifier
STDDEV_SAMP	standard aggregate function, non-reserved identifier
SUBSTRING	function, non-reserved identifier
SUM	standard aggregate function, non-reserved identifier
TEXTAGG	non-reserved identifier, text aggreate function
TEXTTABLE	non-reserved identifier, text table
TIMESTAMPADD	function, non-reserved identifier
TIMESTAMPDIFF	function, non-reserved identifier
TO_BYTES	function, non-reserved identifier
TO_CHARS	function, non-reserved identifier
TRANSLATOR	ALTER DATA WRAPPER, create data wrapper, create server, aka data source, Drop data wrapper, non-reserved identifier
TRIM	function, non-reserved identifier, text table column, text table
TYPE	alter column options, create data wrapper, create server, aka data source, non-reserved identifier
UNBOUNDED	non-reserved identifier, window frame bound
UPSERT	insert statement, non-reserved identifier
USAGE	Create GRANT, non-reserved identifier, Revoke GRANT
USE	non-reserved identifier, use database
VARIADIC	non-reserved identifier, procedure parameter
VAR_POP	standard aggregate function, non-reserved identifier
VAR_SAMP	standard aggregate function, non-reserved identifier
VERSION	create database, create server, aka data source, Import another Database, non-reserved identifier, use database, xml serialize
VIEW	alter, ALTER TABLE, create view, drop table, non-reserved identifier

WELLFORMED	non-reserved identifier, xml parse
WIDTH	non-reserved identifier, text table column
XMLDECLARATION	non-reserved identifier, xml serialize

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

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
comma	","	alter child options list, alter options list, ARRAY expression constructor, column list, create procedure, typed element list, create table body, create temporary table, create view body, derived column list, sql exception, named parameter list, expression list, from clause, function, Create GRANT, limit clause, nested expression, object table, option clause, options clause, order by clause, simple data type, query expression, query primary, querystring function, identifier list, Revoke GRANT, select clause, set clause list, in predicate, text aggreate function, text table, xml attributes, xml element, xml query, xml forest, xml namespaces, xml query, xml table

concat_op	"||"	common value expression
decimal numeric literal	(<digit>)* <period> <unsigned integer literal>	unsigned numeric literal
digit	\["0"\-"9"\]	
dollar	"$"	parameter reference
double_amp_op	"&&"	common value expression
eq	"="	assignment statement, callable statement, declare statement, named parameter list, comparison operator, set clause list
escaped function	"{" "fn"	unsigned value expression primary
escaped join	"{" "oj"	table reference
escaped type	"{" ("d" | "t" | "ts" | "b")	non numeric literal
approximate numeric literal	<digit> <period> <unsigned integer literal> \["e","E"\] (<plus> | <minus>)? <unsigned integer literal>	unsigned numeric literal
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	\["a"\-"z","A"\-"Z"\] | \["\u0153"\-"\ufffd"\]	
		standard aggregate function, alter child options list, alter options list, analytic aggregate function, array table, callable statement, column list, other constraints, create procedure, create table body, create temporary table, create view body, filter clause, function, group by clause, if

lparen	"("	statement, json object, 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
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 table, callable statement, column list, other constraints, create procedure, create table body, create temporary table, create view body, filter clause, function, group by clause, if statement, json object, 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
string literal	("N" | "E")? "\'" ("\\'" | ~\["\'"\])* "\'"	string
unsigned integer literal	(<digit>)+	unsigned integer, unsigned numeric literal

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
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
between predicate	boolean primary
boolean primary	filter clause, boolean factor
branching statement	delimited statement
case expression	unsigned value expression primary
character	match predicate, text aggreate 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
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 role	ddl statement
create schema	ddl statement
create server, aka data source	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
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 aggreate 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 procedure	ddl statement
drop role	ddl statement
drop schema	ddl statement
drop server, aka data source	ddl statement
drop table	directly executable statement
drop table	ddl statement
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
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
Create GRANT	ddl statement
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 schema, create temporary table, create trigger, view element, declare statement, delete statement, derived column, drop option, Drop data wrapper, drop option, drop procedure, drop role, drop schema, drop server, aka data source, drop table, drop table, dynamic data statement, exception reference, named parameter list, foreign key, function, Create GRANT, Import another Database, Import foreign schema, insert statement, into clause, loop statement, xml namespace element, object table column, object table, option clause, option pair, procedure parameter, procedure result column, query primary, identifier list, 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, use schema, with list element, xml table column, xml element, xml serialize, xml table
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
unsigned integer	dynamic data statement, function, Create 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 object	function
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
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, aka data source, 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, 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
qualified table	joined table
query	query primary
query expression	alter, ALTER TABLE, 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
identifier list	create schema, with role
grant type	Create 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
data statement	delimited statement
statement	alter, ALTER PROCEDURE, compound statement, create procedure, for each row trigger action, if statement, loop statement, procedure body definition, while statement
call statement	assignment statement, subquery, table subquery, directly executable statement
string	character, create database, option namespace, create server, aka data source, function, Create GRANT, Import another Database, 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 aggreate 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
Unqualified identifier	create procedure, create data wrapper, create foreign or global temporary table, create role, create server, aka data source, 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
use schema	ddl statement
directly executable statement	data statement
value expression primary	array 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
xml parse	function

xml query	function
xml serialize	function
xml table	table primary

Productions

string ::=

- <string literal>

A string literal value. Use '' to escape ' in the string.

Example:

```
'a string'
```

```
'it''s a string'
```

---

non-reserved identifier ::=

- INSTEAD
- VIEW
- ENABLED
- DISABLED
- KEY
- SERIAL
- 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
- INDEX
- 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
- SELECTOR
- SKIP
- WIDTH
- PASSING
- NAME
- ENCODING
- COLUMNS
- DELIMITER
- QUOTE
- HEADER
- NULLS
- OBJECTTABLE
- VERSION
- INCLUDING
- EXCLUDING
- XMLDECLARATION
- VARIADIC
- RAISE
- EXCEPTION
- CHAIN
- JSON
- JSONARRAY_AGG
- JSONOBJECT
- PRESERVE
- UPSERT
- AFTER
- TYPE
- TRANSLATOR
- JAAS
- CONDITION
- MASK

- ACCESS
- CONTROL
- NONE
- DATA
- DATABASE
- PRIVILEGES
- ROLE
- SCHEMA
- USE
- REPOSITORY
- RENAME
- DOMAIN
- USAGE
- GEOMETRY
- DEFAULT
- POSITION
- CURRENT
- UNBOUNDED
- PRECEDING
- FOLLOWING
- LISTAGG
- GEOGRAPHY

Allows non-reserved keywords to be parsed as identifiers

Example: SELECT **COUNT** FROM …

Unqualified identifier ::=

- <identifier>
- <non-reserved identifier>

Unqualified name of a single entity.

Example:

```
"tbl"
```

---

identifier ::=

- <identifier>
- <non-reserved identifier>

Partial or full name of a single entity.

Example:

```
tbl.col
```

```
"tbl"."col"
```

---

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
```

---

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
```

---

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
```

---

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
```

---

drop table ::=

- DROP TABLE <identifier>

Drop the given table.

Example:

```
DROP TABLE #temp
```

---

create temporary table ::=

- CREATE (LOCAL)? TEMPORARY TABLE <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)
```

---

temporary table element ::=

- <identifier> (<basic data type> | SERIAL) (NOT NULL)?

Defines a temporary table column.

Example:

```
col string NOT NULL
```

---

raise error statement ::=

- ERROR <expression>

Raises an error with the given message.

Example:

```
ERROR 'something went wrong'
```

---

raise statement ::=

- RAISE (SQLWARNING)? <exception reference>

Raises an error or warning with the given message.

Example:

```
RAISE SQLEXCEPTION 'something went wrong'
```

---

exception reference ::=

- <identifier>
- <sql exception>

a reference to an exception

Example:

```
SQLEXCEPTION 'something went wrong' SQLSTATE '00X', 2
```

---

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
```

---

statement ::=

- ((<identifier> <colon>)? (<loop statement> | <while statement> | <compound statement>))

- <if statement> | <delimited statement>

A procedure statement.

Example:

```
IF (x = 5) BEGIN ... END
```

---

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;
```

---

compound statement ::=

- BEGIN ((NOT)? ATOMIC)? (<statement>)* (EXCEPTION <identifier> (<statement>)*)? END

A procedure statement block contained in BEGIN END.

Example:

```
BEGIN NOT ATOMIC ... END
```

---

branching statement ::=

- ((BREAK | CONTINUE) (<identifier>)?)
- (LEAVE <identifier>)

A procedure branching control statement, which typically specifies a label to return control to.

Example:

```
BREAK x
```

---

return statement ::=

- RETURN (<expression>)?

A return statement.

Example:

```
RETURN 1
```

while statement ::=

- WHILE <lparen> <condition> <rparen> <statement>

A procedure while statement that executes until its condition is false.

Example:

```
WHILE (var) BEGIN ... END
```

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
```

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
```

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'
```

assignment statement ::=

- <identifier> <eq> (<assignment statement operand> | (<call statement> ((WITH | WITHOUT) RETURN)?))

Assigns a variable a value in a procedure.

Example:

```
x = 'b'
```

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}

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.

procedure body definition ::=

- (CREATE (VIRTUAL)? PROCEDURE)? <statement>

Defines a procedure body on a Procedure metadata object.

Example:

```
BEGIN ... END
```

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
```

set clause list ::=

- <identifier> <eq> <expression> (<comma> <identifier> <eq> <expression>)*

A list of value assignments.

Example:

```
col1 = 'x', col2 = 'y' ...
```

---

typed element list ::=

- <identifier> <basic data type> (<comma> <identifier> <basic data type>)*

A list of typed elements.

Example:

```
col1 string, col2 integer ...
```

---

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}
```

---

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)
```

---

named parameter list ::=

- (<identifier> <eq> (<gt>)? <expression> (<comma> <identifier> <eq> (<gt>)? <expression>)*)

A list of named parameters.

Example:

```
param1 => 'x', param2 => 1
```

---

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)
```

---

expression list ::=

- <expression> (<comma> <expression>)*

A list of expressions.

Example:

```
col1, 'a', ...
```

---

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
```

---

delete statement ::=

- DELETE FROM <identifier> ((AS)? <identifier>)? (<where clause>)? (<option clause>)?

Delete rows from the given target.

Example:

```
DELETE FROM tbl WHERE col2 = 1
```

---

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
```

---

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)
```

---

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
```

---

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
```

---

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
```

---

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
```

---

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
```

---

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
```

---

select sublist ::=

- <select derived column>
- <all in group>

An element in the select clause

Example:

```
tbl.*
```

```
tbl.col AS x
```

---

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
```

---

derived column ::=

- (<expression> (AS <identifier>)?)

An optionally named expression.

Example:

```
tbl.col AS x
```

---

all in group ::=

- <all in group identifier>

A select sublist that can select all columns from the given group.

Example:

```
tbl.*
```

---

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)
```

---

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)
```

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(*)
```

analytic aggregate function ::=

- (ROW_NUMBER | RANK | DENSE_RANK | PERCENT_RANK | CUME_DIST) <lparen> <rparen>

An analytic aggregate function.

Example:

```
ROW_NUMBER()
```

filter clause ::=

- FILTER <lparen> WHERE <boolean primary> <rparen>

An aggregate filter clause applied prior to accumulating the value.

Example:

```
FILTER (WHERE col1='a')
```

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>
```

table reference ::=

- (<escaped join> <joined table> <rbrace>)

- <joined table>

An optionally escaped joined table.

Example:

```
a
```

```
a inner join b
```

---

joined table ::=

- <table primary> (<cross join> | <qualified table>)*

A table or join.

Example:

```
a
```

```
a inner join b
```

---

cross join ::=

- ((CROSS | UNION) JOIN <table primary>)

A cross join.

Example:

```
a CROSS JOIN b
```

---

qualified table ::=

- (((RIGHT (OUTER)?) | (LEFT (OUTER)?) | (FULL (OUTER)?) | INNER)? JOIN <table reference> ON <condition>)

An INNER or OUTER join.

Example:

```
a inner join b
```

---

table primary ::=

- (<text table> | <array 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
```

---

make dep options ::=

- (<lparen> (MAX <colon> <unsigned integer>)? ((NO)? JOIN)? <rparen>)?

options for the make dep hint

Example:

```
(min:10000)
```

---

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)
```

---

array table ::=

- ARRAYTABLE <lparen> <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
```

---

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
```

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
```

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)
```

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)
```

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
```

object table column ::=

- <identifier> <basic data type> <string> (DEFAULT <expression>)?

object table column.

Example:

```
y integer 'teiid_row_number'
```

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
```

xml table column ::=

- <identifier> ((FOR ORDINALITY) | (<basic data type> (DEFAULT <expression>)? (PATH <string>)?))

XML table column.

Example:

```
y FOR ORDINALITY
```

unsigned integer ::=

- <unsigned integer literal>

An unsigned interger value.

Example:

```
12345
```

table subquery ::=

- (TABLE | LATERAL)? <lparen> (<query expression> | <call statement>) <rparen> (AS)? <identifier>

A table defined by a subquery.

Example:

```
(SELECT * FROM tbl) AS x
```

table name ::=

- (<identifier> ((AS)? <identifier>)?)

A table named in the FROM clause.

Example:

```
tbl AS x
```

---

where clause ::=

- WHERE <condition>

Specifies a search condition

Example:

```
WHERE x = 'a'
```

---

condition ::=

- <boolean value expression>

A boolean expression.

---

boolean value expression ::=

- <boolean term> (OR <boolean term>)*

An optionally ORed boolean expression.

---

boolean term ::=

- <boolean factor> (AND <boolean factor>)*

An optional ANDed boolean factor.

---

boolean factor ::=

- (NOT)? <boolean primary>

A boolean factor.

Example:

```
NOT x = 'a'
```

---

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%'
```

---

comparison operator ::=

- <eq>
- <ne>
- <ne2>
- <lt>
- <le>
- <gt>
- <ge>

A comparison operator.

Example:

```
=
```

---

is distinct ::=

- IS (NOT)? DISTINCT FROM <common value expression>

Is Distinct Right Hand Side

Example:

```
IS DISTINCT FROM expression
```

---

comparison predicate ::=

- <comparison operator> <common value expression>

A value comparison.

Example:

```
= 'a'
```

subquery ::=

- <lparen> (<query expression> | <call statement>) <rparen>

A subquery.

Example:

```
(SELECT * FROM tbl)
```

quantified comparison predicate ::=

- <comparison operator> (ANY | SOME | ALL) (<subquery> | (<lparen> <expression> <rparen>))

A subquery comparison.

Example:

```
= ANY (SELECT col FROM tbl)
```

match predicate ::=

- (NOT)? (LIKE | (SIMILAR TO)) <common value expression> (ESCAPE <character> | (<lbrace> ESCAPE <character> <rbrace>))?

Matches based upon a pattern.

Example:

```
LIKE 'a_'
```

like regex predicate ::=

- (NOT)? LIKE_REGEX <common value expression>

A regular expression match.

Example:

```
LIKE_REGEX 'a.*b'
```

character ::=

- <string>

A single character.

Example:

```
'a'
```

---

between predicate ::=

- (NOT)? BETWEEN <common value expression> AND <common value expression>

A comparison between two values.

Example:

```
BETWEEN 1 AND 5
```

---

is null predicate ::=

- IS (NOT)? NULL

A null test.

Example:

```
IS NOT NULL
```

---

in predicate ::=

- (NOT)? IN (<subquery> | (<lparen> <common value expression> (<comma> <common value expression>)* <rparen>))

A comparison with multiple values.

Example:

```
IN (1, 5)
```

---

exists predicate ::=

- EXISTS <subquery>

A test if rows exist.

Example:

```
EXISTS (SELECT col FROM tbl)
```

---

group by clause ::=

- GROUP BY (ROLLUP <lparen> <expression list> <rparen> | <expression list>)

Defines the grouping columns

Example:

```
GROUP BY col1, col2
```

---

having clause ::=

- HAVING <condition>

Search condition applied after grouping.

Example:

```
HAVING max(col1) = 5
```

---

order by clause ::=

- ORDER BY <sort specification> (<comma> <sort specification>)*

Specifices row ordering.

Example:

```
ORDER BY x, y DESC
```

---

sort specification ::=

- <sort key> (ASC | DESC)? (NULLS (FIRST | LAST))?

Defines how to sort on a particular expression

Example:

```
col1 NULLS FIRST
```

---

sort key ::=

- <expression>

A sort expression.

Example:

```
col1
```

---

integer parameter ::=

- <unsigned integer>

- <unsigned value expression primary>

A literal integer or parameter reference to an integer.

Example:

```
?
```

---

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
```

---

fetch clause ::=

- FETCH (FIRST | NEXT) (<integer parameter>)? (ROW | ROWS) ONLY

ANSI limit.

Example:

```
FETCH FIRST 1 ROWS ONLY
```

---

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
```

---

expression ::=

- <condition>

A value.

Example:

```
col1
```

---

common value expression ::=

- (<numeric value expression> ((<double_amp_op> | <concat_op>) <numeric value expression>)*)

Establishes the precedence of concat.

Example:

```
'a' || 'b'
```

---

numeric value expression ::=

- (<term> (<plus or minus> <term>)*)

Example:

```
1 + 2
```

---

plus or minus ::=

- <plus>
- <minus>

The + or - operator.

Example:

```
+
```

---

term ::=

- (<value expression primary> (<star or slash> <value expression primary>)*)

A numeric term

Example:

```
1 * 2
```

---

star or slash ::=

- <star>
- <slash>

The * or / operator.

Example:

```
/
```

---

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
```

---

parameter reference ::=

- <qmark>
- (<dollar> <unsigned integer>)

A parameter reference to be bound later.

Example:

```
?
```

---

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>)

---

nested expression ::=

- (<lparen> (<expression> (<comma> <expression>)*)? (<comma>)? <rparen>)

An expression nested in parens

Example:

```
(1)
```

---

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
```

ARRAY expression constructor ::=

- (ARRAY <lsbrace> (<expression> (<comma> <expression>)*)? <rsbrace>)

Creates and array of the given expressions.

Example:

```
----ARRAY[1,2]
----
```

---

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)
```

---

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
```

---

window frame bound ::=

- ((UNBOUNDED | <unsigned integer>) (FOLLOWING | PRECEDING))
- (CURRENT ROW)

Defines the start or end of a window frame

Example:

```
CURRENT ROW
```

---

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
```

---

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
```

---

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) 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)
```

---

xml parse ::=

- XMLPARSE <lparen> (DOCUMENT | CONTENT) <expression> (WELLFORMED)? <rparen>

Parses the given value as XML.

Example:

```
XMLPARSE(DOCUMENT doc WELLFORMED)
```

---

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)
```

---

xml element ::=

- XMLELEMENT <lparen> ((NAME)? <identifier>) (<comma> <xml namespaces>)? (<comma> <xml attributes>)? (<comma> <expression>)* <rparen>

Creates an XML element.

Example:

```
XMLELEMENT(NAME "root", child)
```

---

xml attributes ::=

- XMLATTRIBUTES <lparen> <derived column> (<comma> <derived column>)* <rparen>

Creates attributes for the containing element.

Example:

```
XMLATTRIBUTES(col1 AS attr1, col2 AS attr2)
```

---

json object ::=

- JSONOBJECT <lparen> <derived column list> <rparen>

Produces a JSON object containing name value pairs.

Example:

```
JSONOBJECT(col1 AS val1, col2 AS val2)
```

---

derived column list ::=

- <derived column> (<comma> <derived column>)*

a list of name value pairs

Example:

```
col1 AS val1, col2 AS val2
```

---

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)
```

---

xml namespaces ::=

- XMLNAMESPACES <lparen> <xml namespace element> (<comma> <xml namespace element>)* <rparen>

Defines XML namespace URI/prefix combinations

Example:

```
XMLNAMESPACES('http://foo' AS foo)
```

---

xml namespace element ::=

- (<string> AS <identifier>)
- (NO DEFAULT)
- (DEFAULT <string>)

An xml namespace

Example:

```
NO DEFAULT
```

---

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
- (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
```

---

basic data type ::=

- <simple data type> (<lsbrace> <rsbrace>)*

A data type.

Example:

```
STRING[]
```

---

data type ::=

- <basic data type>
- (<identifier> (<lsbrace> <rsbrace>)*)

A data type.

Example:

```
STRING[]
```

---

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
```

---

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'
```

---

unsigned numeric literal ::=

- <unsigned integer literal>
- <approximate numeric literal>
- <decimal numeric literal>

An unsigned numeric literal value.

Example:

```
1.234
```

---

ddl statement ::=

- <create table> (<create table> | <create procedure>)?
- <option namespace>
- <alterStatement>
- <create trigger>
- <create a domain or type alias>
- <create server, aka data source>
- <create role>
- <drop role>
- <Create GRANT>
- <Revoke GRANT>
- <drop server, aka data source>
- <drop table>
- <Import foreign schema>
- <Import another Database>
- <create database>
- <use database>
- <drop schema>
- <use 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)
```

---

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
```

---

create database ::=

- CREATE DATABASE <identifier> (VERSION <string>)? (<options clause>)?

create a new database

Example:

```
CREATE DATABASE foo [VERSION 'version'] OPTIONS(...)
```

---

use database ::=

- USE DATABASE <identifier> (VERSION <string>)?

database into working context

Example:

```
USE DATABASE foo [VERSION 'version']
```

---

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) OPTIONS(...)
```

---

drop schema ::=

- DROP (VIRTUAL | FOREIGN)? SCHEMA <identifier>

drop a schema in database

Example:

```
----DROP SCHEMA foo
----
```

---

use schema ::=

- SET SCHEMA <identifier>

use schema for following database resources

Example:

```
USE SCHEMA foo
```

---

create a domain or type alias ::=

- CREATE DOMAIN <identifier> (AS)? <data type> (NOT NULL)?

creates a named type with optional constraints

Example:

```
CREATE DOMAIN my_type AS INTEGER NOT NULL
```

---

create data wrapper ::=

- CREATE FOREIGN (DATA WRAPPER | TRANSLATOR) <Unqualified identifier> (TYPE <identifier>)? (<options clause>)?

Defines a translator; use the options to override the translator properties.

Example:

```
CREATE FOREIGN (DATA WRAPPER|TRANSLATOR)  wrapper OPTIONS(properties)
```

Drop data wrapper ::=

- DROP FOREIGN (DATA WRAPPER | TRANSLATOR) <identifier>

Deletes a translator

Example:

```
DROP FOREIGN (DATA WRAPPER|TRANSLATOR) wrapper
```

---

create role ::=

- CREATE ROLE <Unqualified identifier> (WITH <with role>)?

Defines data role for the database

Example:

```
CREATE DATA ROLE <data-role>  [WITH JAAS ROLE <string>(,<string>)*]
```

with role ::=

- (JAAS ROLE <identifier list> | ANY AUTHENTICATED) (WITH (JAAS ROLE <identifier list> | ANY AUTHENTICATED))*

drop role ::=

- DROP ROLE <identifier>

Removes data role for the database

Example:

```
DROP ROLE <data-role>
```

Create GRANT ::=

- GRANT (((<grant type> (<comma> <grant type>)*)? ON (TABLE <identifier> (CONDITION (CONSTRAINT)? <string>)? | FUNCTION <identifier> | PROCEDURE <identifier> (CONDITION (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
```

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
```

create server, aka data source ::=

- CREATE SERVER <Unqualified identifier> (TYPE <string>)? (VERSION <string>)? FOREIGN (DATA WRAPPER | TRANSLATOR) <Unqualified identifier> (<options clause>)?

Defines connection to foreign source

Example:

```
CREATE SERVER server_name [ TYPE 'server_type' ] [ VERSION 'server_version' ] FOREIGN (<DATA> <WRAPPER>|<TRANSL
ATOR>)  fdw_name [ OPTIONS ( option 'value' [, ... ] ) ]
```

---

drop server, aka data source ::=

- DROP SERVER <identifier>

Defines dropping connection to foreign source

Example:

```
----DROP SERVER server_name
----
```

---

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
```

---

drop procedure ::=

- DROP (VIRTUAL | FOREIGN)? (PROCEDURE | FUNCTION) <identifier>

Drops a table or view.

Example:

```
DROP [FOREIGN (TABLE|VIEW) table-name
```

---

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
```

---

procedure result column ::=

- <identifier> <data type> (NOT NULL)? (<options clause>)?

A procedure result column.

Example:

```
x INTEGER
```

---

create table ::=

- CREATE (<create view> | <create foreign or global temporary table>)

Defines a table or view.

Example:

```
CREATE VIEW vw AS SELECT 1
```

---

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)
```

---

create view ::=

- (VIRTUAL)? VIEW <Unqualified identifier> (<create view body> | (<options clause>)?) AS <query expression>

Defines a view.

Example:

```
VIEW vw AS SELECT 1
```

---

drop table ::=

- DROP ((FOREIGN TABLE) | ((VIRTUAL)? VIEW) | (GLOBAL TEMPORARY TABLE)) <identifier>

Drops a table or view.

Example:

```
DROP (FOREIGN TABLE | [VIRTUAL] VIEW) table-name
```

---

create foreign temp table ::=

- CREATE (LOCAL)? FOREIGN TEMPORARY TABLE <identifier> <create table body> ON <identifier>

Defines a foreign temp table

Example:

```
CREATE FOREIGN TEMPORARY TABLE t (x string) ON z
```

---

create table body ::=

- <lparen> <table element> (<comma> (<table constraint> | <table element>))* <rparen> (<options clause>)?

Defines a table.

Example:

```
(x string) OPTIONS (CARDINALITY 100)
```

---

create view body ::=

- <lparen> <view element> (<comma> (<table constraint> | <view element>))* <rparen> (<options clause>)?

Defines a view.

Example:

```
(x) OPTIONS (CARDINALITY 100)
```

---

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)
```

---

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)
```

---

primary key ::=

- PRIMARY KEY <column list>

Defines the primary key.

Example:

```
PRIMARY KEY (a, b)
```

---

other constraints ::=

- ((UNIQUE | ACCESSPATTERN) <column list>)
- (INDEX <lparen> <expression list> <rparen>)

Defines ACCESSPATTERN and UNIQUE constraints and INDEXes.

Example:

```
UNIQUE (a)
```

---

column list ::=

- <lparen> <identifier> (<comma> <identifier>)* <rparen>

A list of column names.

Example:

```
(a, b)
```

---

table element ::=

- <identifier> (SERIAL | (<data type> (NOT NULL)? (AUTO_INCREMENT)?)) <post create column>

Defines a table column.

Example:

```
x INTEGER NOT NULL
```

---

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
```

post create column ::=

- (<inline constraint>)? (DEFAULT <expression>)? (<options clause>)?

Common options trailing a column

Example:

```
PRIMARY KEY
```

inline constraint ::=

- (PRIMARY KEY)
- UNIQUE
- INDEX

Defines a constraint on a single column

Example:

```
x INTEGER PRIMARY KEY
```

options clause ::=

- OPTIONS <lparen> <option pair> (<comma> <option pair>)* <rparen>

A list of statement options.

Example:

```
OPTIONS ('x' 'y', 'a' 'b')
```

option pair ::=

- <identifier> (<non numeric literal> | (<plus or minus>)? <unsigned numeric literal>)

An option key/value pair.

Example:

```
'key' 'value'
```

alter option pair ::=

- <identifier> (<non numeric literal> | (<plus or minus>)? <unsigned numeric literal>)

Alter An option key/value pair.

Example:

```
'key' 'value'
```

---

alterStatement ::=

- ALTER (<ALTER TABLE> | <ALTER PROCEDURE> | <ALTER TRIGGER> | <ALTER SERVER> | <ALTER DATA WRAPPER> | <ALTER DATABASE>)

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|DROP|ALTER) COLUMN <name> <type> OPTIONS ( (ADD|SET|DROP) x y)
```

RENAME Table ::=

- TO <identifier>

alters table name

Example:

```
ALTER TABLE foo RENAME TO BAR;
```

ADD constraint ::=

- ADD <table constraint>

alters table and adds a constraint

Example:

```
ADD PRIMARY KEY (ID)
```

ADD column ::=

- ADD COLUMN <table element>

alters table and adds a column

Example:

```
ADD COLUMN bar type OPTIONS (ADD updatable true)
```

DROP column ::=

- DROP COLUMN <identifier>

alters table and adds a column

Example:

```
----DROP COLUMN bar
----
```

---

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)
```

---

rename column options ::=

- <identifier> TO <identifier>

renames either a table column or procedure's parameter name

Example:

```
RENAME COLUMN bar TO foo
```

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 [AS <stmt>] OPTIONS (ADD x y)
```

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 <id> INSTEAD OF (INSERT|UPDATE|DELETE) AS [ENABLED|DISABLED]
```

ALTER SERVER ::=

- SERVER <identifier> <alter options list>

alters options of database

Example:

```
ALTER SERVER foo OPTIONS (ADD x y)
```

ALTER DATA WRAPPER ::=

- (DATA WRAPPER | TRANSLATOR) <identifier> <alter options list>

alters options of data wrapper

Example:

```
ALTER [DATA WRAPPER|TRANSLATOR] foo OPTIONS (ADD x y)
```

ALTER DATABASE ::=

- DATABASE <identifier> <alter options list>

alters options of database

Example:

```
ALTER DATABASE foo OPTIONS (ADD x y)
```

---

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)
```

---

drop option ::=

- DROP <identifier>

drop option

Example:

```
DROP updatable
```

---

add set option ::=

- (ADD | SET) <alter option pair>

add or set an option pair

Example:

```
ADD updatable true
```

---

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)
```

---

drop option ::=

- DROP <identifier>

drop option

Example:

```
DROP updatable
```

---

add set child option ::=

- (ADD | SET) <alter child option pair>

add or set an option pair

Example:

```
ADD updatable true
```

alter child option pair ::=

- <identifier> (<non numeric literal> | (<plus or minus>)? <unsigned numeric literal>)

Alter An option key/value pair.

Example:

```
'key' 'value'
```

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 [LIMIT TO (x,y,z)|EXCEPT (x,y,z)] FROM SERVER bar
```

Import another Database ::=

- IMPORT DATABASE <identifier> VERSION <string> (WITH ACCESS CONTROL)?

imports another database into current database

Example:

```
IMPORT DATABASE <id> VERSION <string-val> [WITH ACCESS CONTROL]
```

identifier list ::=

- <identifier> (<comma> <identifier>)*

grant type ::=

- SELECT
- INSERT
- UPDATE
- DELETE
- EXECUTE
- ALTER
- DROP