Table of Contents

Legal Notice

Teiid 14.0 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:

```
$ cd wildfly
$ 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

Administrator’s Guide

This guide is intended for any user who assumes role of a developer/administrator of a Teiid instance.

This guide guides user through installation of Teiid, configuration of different services and deployment of Teiid artifacts such as VDBs.

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.

Installation Guide

Teiid needs to be installed into an existing WildFly 19.1.0 installation.

Note	Teiid provides an embedded kit, however it should be considered a tech preview as its APIs will likely evolve and there is sparse documentation.

Steps to install Teiid

- Download the WildFly application server. Install the server by unzipping into a known location. Ex: /apps/jboss-install

Note	You may also choose to use an existing AS installation. However if a previous version of Teiid was already installed, you must remove the old Teiid distribution artifacts before installing the new version.

- Download Teiid. Unzip the downloaded artifact inside the WildFly installation. Teiid 14.0 directory structure matches WildFly directly - it is just an overlay. This will add necessary modules and configuration files to install Teiid in WildFly 19.1.0 in both *Standalone* and *Domain* modes. Teiid provides separate configuration files for both standalone mode and domain mode. Based on mode type you selected to run WildFly 19.1.0 , you may have to run a CLI script to complete the Teiid installation.

The "Domain" mode recommended in a clustered environment to take advantage of clustered caching and cluster safe distribution of events. Teiid's default configuration for Domain mode through CLI script configured for high availability and clustered caching.

Standalone Mode

if you want to start the "standalone" profile, execute the following command

```
<jboss-install>/bin/standalone.sh -c=standalone-teiid.xml
```

Installing Teiid using CLI script

The above is starting WildFly in a separate Teiid specific configuration that is based standalone.xml. However, if you already working with a predefined configuration for example default *standalone.xml* and would like to install Teiid into that configuration, then you can execute the following JBoss CLI script. First, start the server

```
<jboss-install>/bin/standalone.sh
```

then in a separate console window execute

```
<jboss-install>/bin/jboss-cli.sh --file=bin/scripts/teiid-standalone-mode-
install.cli
```

this will install Teiid subsystem into the running configuration of the WildFly 19.1.0 in standalone mode.

Note: If you are using standalone ha or standalone full-ha, you should use the teiid-standalone-ha-mode-install.cli script instead.

Domain Mode

To start the server in "Domain" mode, install WildFly 19.1.0 and Teiid 14.0 on all the servers that are going to be part of the cluster. Select one of the servers as the "master" domain controller, the rest of the servers will be slaves that connect to the "master" domain controller for all the administrative operations. Please refer to WildFly 19.1.0 provided documentation for full details.

Once you configured all the servers, start the "master" node with following command

```
<jboss-install>/bin/domain.sh
```

and on "slave" nodes

```
<jboss-install>/bin/domain.sh
```

The slave nodes fetch their domain configuration from the "master" node.

Once all the servers are up, complete the installation to run in domain mode by executing the following command against the "master" node. Note that this only needs to be run once per domain (i.e. cluster) install. This script will install Teiid in the **ha** and **full-ha** profiles. It will also re-configure *main-server-group* to start the *ha* profile. Once in domain mode, you can not statically deploy resources by dropping them in the *domain/deployments* folder, so this script will deploy the default resources (file, ldap, salesforce and ws connectors) using the CLI interface.

```
<jboss-install>/bin/jboss-cli.sh --file=bin/scripts/teiid-domain-mode-install.cli
```

Thats it!. WildFly and Teiid are now installed and running. See below instructions to customize various other settings.

Once VDBs have been deployed, users can now connect their JDBC applications to Teiid. If you need help on connecting your application to Teiid using JDBC check out the Client Developer’s Guide.

Directory Structure Explained

This shows the contents of the Teiid 14.0 deployment. The directory structure is exactly the same under any JBoss profile.

Directory Structure

```
/bin
   /scripts
/docs
   /teiid
       /datsources
       /schema
       /examples
/domain
   /configuration
/modules
   /system
       /layers
           /base
               /org/jboss/teiid/*
/standalone
   /configuration
      standalone-teiid.xml
```

Name	Description
bin/scripts	Contains installation and utility CLI scripts for setting up Teiid in different configurations.
docs/teiid	Contains documents, examples, sample data source XML fragments and schema files.
/standalone/configuration	standalone-teiid.xml - Master configuration file for the Teiid system. This file contains the Teiid subsystem, in addition to the standard WildFly web profile subsystems
/domain/configuration/	-
/modules/system/layers/base/org/jboss/teiid/*	This directory contains the Teiid modules for WildFly 19.1.0 system
/modules/system/layers/base/org/jboss/teiid/client	This directory contains Teiid client libraries. It has the Teiid JDBC driver jar, "teiid-14.0.0-jdbc.jar", and also contains "teiid-hibernate-dialect-14.0.0.jar" that contains Teiid’s Hibernate dialect.
{standalone or domain}/tmp/teiid	This directory under standalone or domain, contains temporary files created by Teiid. These are mostly created by the buffer manager. These files are not needed across a VM restart. Creation of Teiid lob values(for example through SQL/XML) will typically create one file per lob once it exceeds the allowable in memory size of 8KB. In heavy usage scenarios, consider pointing the buffer directory at a partition that is routinely defragmented.
{standalone or domain}/data/teiid-data	This directory under standalone or domain, contains cached vdb metadata files. Do not edit them manually.

Dockerize Teiid

Running Teiid as a Docker container is straight-forward, but since the runtime by itself is not a turn-key environment you must consider how you will configure/use the server from there.

The following is a basic Dockerfile that can be used to create a base image. Just create a Dockerfile with these contents and run "docker build ." from that directory.

```
FROM jboss/wildfly:19.1.0.Final

ENV JBOSS_HOME /opt/jboss/wildfly

# Set the TEIID_VERSION env variable
ENV TEIID_VERSION 14.0.0

# Download and unzip Teiid server
RUN cd $JBOSS_HOME \
    && curl -O
https://oss.sonatype.org/service/local/repositories/releases/content/org/teiid/teii
d/$TEIID_VERSION/teiid-wildfly-$TEIID_VERSION-dist.zip \
    && bsdtar -xf teiid-wildfly-$TEIID_VERSION-dist.zip \
    && chmod +x $JBOSS_HOME/bin/*.sh \
    && rm teiid-wildfly-$TEIID_VERSION-dist.zip

VOLUME ["$JBOSS_HOME/standalone", "$JBOSS_HOME/domain"]

USER jboss

ENV LAUNCH_JBOSS_IN_BACKGROUND true

# Expose Teiid server  ports
EXPOSE 8080 9990 31000 35432

# Run Teiid server and bind to all interface
CMD ["/bin/sh", "-c", "$JBOSS_HOME/bin/standalone.sh -c standalone-teiid.xml -b
0.0.0.0 -bmanagement 0.0.0.0"]
```

Pre-built images can be found at Docker Hub.

If you are just using the Teiid Docker environment for more than just testing you will likely want to extend the base image or base Dockerfile to overlay the necessary modules, vdbs, and other artifacts as well as run any necessary cli to create your data sources.

Mutable Container

See the Teiid Docker Quickstart that shows starting the Teiid container and performing mutative operations after it is started.

Immutable Container

See the WildFly with MySQL example that shows extending the WildFly image to include a MySQL source. Note that this is based upon also having the database containerized and thus exposing the container linking variables. If that is not the case for your environment, you will have to provide the host/port information in a different way.

OpenShift

OpenShift is the Red Hat enterprise offering of Kubernetes which also utilizes Docker. While you may usually be able to use your existing Docker containers on OpenShift, there are additional considerations and features.

See JDV on OpenShift for an overview of how the productized version of Teiid can be run on OpenShift.

A simplified form of immutable containers, but with additional OpenShift features such as health checks and better JVM constraints, can be seen at OpenShift Teiid Server Docker. Note however that many of the resource concerns have been addressed by later java versions which automatically detect memory constraints in vm sizing and report an appropriate number of available processors.

See link:http://teiid.io/tools/beetle_studio/ for tooling that provides a turn-key experience for creating containerized virtualizations. Under the covers it uses fabric8 and Thorntail to create images. See Teiid Thorntail Examples for direct usage of the build logic.

Deploying VDBs

A VDB is the primary means to define a Virtual Database in Teiid. See the Reference Guide to create a VDB.

Once you have a "VDB" built it can be deployed/undeployed in Teiid runtime in different ways.

Warning	If VDB versioning is not used to give distinct version numbers, overwriting a VDB of the same name will terminate all connections to the old VDB. It is recommended that VDB versioning be used for production systems.

Caution	Removing an existing VDB will immediately clean up VDB file resources, and will automatically terminate existing sessions.

Caution	The runtime names of deployed VDB artifacts must either be *.vdb for a zip file or *-vdb.xml for an xml file or -vdb.ddl for DDL 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.

Tip	if you have existing VDB in combination of *.vdb or -vdb.xml 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.

Direct File Deployment

Copy the VDB file into the

```
<jboss-install>/standalone/deployments
```

directory. Then create an empty marker file with same name as the VDB with extension ".dodeploy" in the same directory. For example, if your vdb name is "enterprise.vdb", then marker file name must be "enterprise.vdb.dodeploy". Make sure that there are no other VDB files with the same name. If a VDB already exists with the same name, then this VDB will be replaced with the new VDB. This is the simplest way to deploy a VDB. This is mostly designed for quick deployment during development, when the Teiid server is available locally on the developer’s machine.

Note	This only works in the Standalone mode. For Domain mode, you must use one of the other available methods.

Admin Console Deployment (Web)

Use the admin web console at:

```
http://<host>:<port>/console
```

More details for this can be found in the Admin Console VDB deployment section. This is the easiest way to deploy a VDB to a remote server.

CLI based Deployment

WildFly 19.1.0 provides command line interface (CLI) for doing any kind of administrative task. Execute

```
bin/jboss-cli.sh --connect
```

command and run

```
# in stand alone mode
deploy /path/to/my.vdb

# in domain mode
deploy /path/to/my.vdb --server-groups=main-server-group
```

to deploy the VDB. Note that in domain mode, you need to either select a particular "server-group" or all available server groups are deployment options. Check out CLI documentation for more general usage of the CLI.

AdminAPI Deployment

See the "deploy" method. Consult the AdminAPI documentation for more information. When using AdminAPI, in domain mode, the VDB is deployed to all the available servers.

Admin API Deployment

The Admin API (look in org.teiid.adminpi.*) provides Java API methods that lets a user connect to a Teiid runtime and deploy a VDB. If you need to programatically deploy a VDB use this method. This method is preferable for OEM users, who are trying to extend the Teiid's capabilities through their applications. When using Admin API, in domain mode, the VDB is deployed to all the servers.

Deploying VDB Dependencies

Apart from deploying the VDB, the user is also responsible for providing all the necessary dependent libraries, configuration for creating the data sources that are needed by the models (schemas) defined in "META-INF/vdb.xml" file inside your VDB. For example, if you are trying to integrate data from Oracle RDBMS and File sources in your VDB, then you are responsible for providing the JDBC driver for the Oracle source and any necessary documents and configuration that are needed by the File Translator.

Data source instances may be used by single VDB, or may be shared with as other VDBs or other applications. Consider sharing connections to data sources that have heavy-weight and resource constrained.

With the exception of JDBC sources, other supported data sources have a corresponding JCA connector in the Teiid kit. Either directly edit the standalone-teiid.xml or use CLI to create the required data sources by the VDB. Example configurations are provided for all the sources in "<jboss-install>/docs/teiid/datasources" directory. Note that in the *Domain* mode, you must use CLI or admin-console or AdminAPI to configure the data sources.

Some data sources may contain passwords or other sensitive information. See the WIKI article EncryptingDataSourcePasswords to not store passwords in plain text.

Once the VDB and its dependencies are deployed, then client applications can connect using the JDBC API. If there are any errors in the deployment, a connection attempt will not be successful and a message will be logged. You can use the admin-console tool or check the log files for errors and correct them before proceeding. Check Client Developer’s Guide on how to use JDBC to connect to your VDB.

Apache Accumulo Data Sources

Accumulo data sources use a Teiid specific JCA connector that is deployed into WildFly 19.1.0 during installation. There are many ways to create a Accumulo data source, using CLI, admin-console, etc. The example shown below uses the CLI tool, as this works in both Standalone and Domain modes.

Execute the following command using the CLI once you connected to the Server. Make sure you provide the correct URL and user credentials. Add any additional properties required by the connector by duplicating the "connection-definitions" command below. Edit the JNDI name to match the JNDI name you used in VDB.

```
batch
/subsystem=resource-adapters/resource-adapter=accumulo/connection-definitions=teiid:add(jndi-name=java:/accumulo-ds, class-name=org.teiid.resource.adapter.accumulo.AccumuloManagedConnectionFactory, enabled=true, use-java-context=true)
/subsystem=resource-adapters/resource-adapter=accumulo/connection-definitions=teiid/config-properties=ZooKeeperServerList:add(value=localhost:2181)
/subsystem=resource-adapters/resource-adapter=accumulo/connection-definitions=teiid/config-properties=Username:add(value=user)
/subsystem=resource-adapters/resource-adapter=accumulo/connection-definitions=teiid/config-properties=Password:add(value=password)
/subsystem=resource-adapters/resource-adapter=accumulo/connection-definitions=teiid/config-properties=InstanceName:add(value=instancename)
/subsystem=resource-adapters/resource-adapter=accumulo/connection-definitions=teiid/config-properties=Roles:add(value=public)
/subsystem=resource-adapters/resource-adapter=accumulo:activate
runbatch
```

All the properties that are defined on the RAR file are

Property Name	Description	Required	Default
ZooKeeperServerList	A comma separated list of zoo keeper server locations. Each location can contain an optional port, of the format host:port	true	none
Username	Connection User’s Name	true	none
Password	Connection User’s password	true	none
InstanceName	Accumulo instance name	true	none
Roles	optional visibility for user, supply multiple with comma separated	false	none

To find out all the properties that are supported by this Accumulo Connector execute the following command in the CLI.

```
/subsystem=teiid:read-rar-description(rar-name=accumulo)
```

Tip	**Developer’s Tip** - If WildFly 19.1.0 is running in standalone mode, you can also manually edit the "<jboss-install>/standalone/configuration/standalone-teiid.xml" file and add the XML configuration defined in **"<jboss-install>/docs/teiid/datasources/accumulo"** directory under "resource-adapters" subsystem. Shutdown the server

	before you edit this file, and restart after the modifications are done.

Amazon SimpleDB Data Sources

SimpleDB data sources use a Teiid specific JCA connector that is deployed into WildFly 19.1.0 during installation. There are many ways to create a SimpleDB data source, using CLI, admin-console, etc. The example shown below uses the CLI tool, as this works in both Standalone and Domain modes.

Execute the following command using the CLI once you connected to the Server. Make sure you provide the correct access keys. Add any additional properties required by the connector by duplicating the "connection-definitions" command below. Edit the JNDI name to match the JNDI name you used in VDB.

```
batch
/subsystem=resource-adapters/resource-adapter=simpledb/connection-definitions=simpledbDS:add(jndi-name=java:/simpledbDS, class-name=org.teiid.resource.adapter.simpledb.SimpleDBManagedConnectionFactory, enabled=true, use-java-context=true)
/subsystem=resource-adapters/resource-adapter=simpledb/connection-definitions=simpledbDS/config-properties=AccessKey:add(value=xxx)
/subsystem=resource-adapters/resource-adapter=simpledb/connection-definitions=simpledbDS/config-properties=SecretAccessKey:add(value=xxx)
/subsystem=resource-adapters/resource-adapter=simpledb:activate
runbatch
```

To find out all the properties that are supported by this SimpleDB Connector execute the following command in the CLI.

```
/subsystem=teiid:read-rar-description(rar-name=simpledb)
```

Tip	**Developer’s Tip** - If WildFly 19.1.0 is running in standalone mode, you can also manually edit the "<jboss-install>/standalone/configuration/standalone-teiid.xml" file and add the XML configuration defined in "<jboss-install>/docs/teiid/datasources/simpledb" directory under "resource-adapters" subsystem. Shutdown the server before you edit this file, and restart after the modifications are done.

Cassandra Data Sources

Cassandra data sources use a Teiid specific JCA connector that is deployed into WildFly 19.1.0 during installation. There are many ways to create a Cassandra data source, using CLI, admin-console, etc. The example shown below uses the CLI tool, as this works in both Standalone and Domain modes.

Execute the following command using the CLI once you connected to the Server. Make sure you provide the correct URL and user credentials. Add any additional properties required by the connector by duplicating the "connection-definitions" command below. Edit the JNDI name to match the JNDI name you used in VDB.

```
batch
/subsystem=resource-adapters/resource-adapter=cassandra/connection-definitions=cassandraDS:add(jndi-name=java:/
cassandraDS, class-name=org.teiid.resource.adapter.cassandra.CassandraManagedConnectionFactory, enabled=true, u
se-java-context=true)
/subsystem=resource-adapters/resource-adapter=cassandra/connection-definitions=cassandraDS/config-properties=Ad
dress:add(value=127.0.0.1)
/subsystem=resource-adapters/resource-adapter=cassandra/connection-definitions=cassandraDS/config-properties=Ke
yspace:add(value=my-keyspace)
/subsystem=resource-adapters/resource-adapter=cassandra:activate
runbatch
```

To find out all the properties that are supported by this Cassandra Connector execute the following command in the CLI.

```
/subsystem=teiid:read-rar-description(rar-name=cassandra)
```

Tip	**Developer’s Tip** - If WildFly 19.1.0 is running in standalone mode, you can also manually edit the "<jboss-install>/standalone/configuration/standalone-teiid.xml" file and add the XML configuration defined in "<jboss-install>/docs/teiid/datasources/cassandra" directory under "resource-adapters" subsystem. Shutdown the server before you edit this file, and restart after the modifications are done.

Couchbase Data Sources

Couchbase data sources use a Teiid specific JCA connector that is deployed into WildFly 19.1.0 during installation. There are many ways to create a Couchbase data source, using CLI, admin-console, etc. The example shown below uses the CLI tool, as this works in both Standalone and Domain modes.

Execute the following command using the CLI once you connected to the Server. Make sure you provide the correct URL and user credentials. Add any additional properties required by the connector by duplicating the "connection-definitions" command below. Edit the JNDI name to match the JNDI name you used in VDB.

```
batch
/subsystem=resource-adapters/resource-adapter=couchbaseQS:add(module=org.jboss.teiid.resource-adapter.couchbase
)
/subsystem=resource-adapters/resource-adapter=couchbaseQS/connection-definitions=couchbaseDS:add(jndi-name="jav
a:/couchbaseDS", class-name=org.teiid.resource.adapter.couchbase.CouchbaseManagedConnectionFactory, enabled=true
, use-java-context=true)
/subsystem=resource-adapters/resource-adapter=couchbaseQS/connection-definitions=couchbaseDS/config-properties=
ConnectionString:add(value="localhost")
/subsystem=resource-adapters/resource-adapter=couchbaseQS/connection-definitions=couchbaseDS/config-properties=
Keyspace:add(value="default")
/subsystem=resource-adapters/resource-adapter=couchbaseQS/connection-definitions=couchbaseDS/config-properties=
Namespace:add(value="default")
runbatch
```

To find out all the properties that are supported by this Couchbase Connector execute the following command in the CLI.

```
/subsystem=teiid:read-rar-description(rar-name=couchbase)
```

Tip	**Developer’s Tip** - If WildFly 19.1.0 is running in standalone mode, you can also manually edit the "<jboss-install>/standalone/configuration/standalone-teiid.xml" file and add the XML configuration defined in "<jboss-install>/docs/teiid/datasources/couchbase" directory under "resource-adapters" subsystem. Shutdown the server before you edit this file, and restart after the modifications are done.

File Data Sources

File data sources use a Teiid specific JCA connector that is deployed into WildFly 19.1.0 during installation. There are many ways to create the file data source, using CLI, admin-console, etc. The example shown below uses the CLI tool, as this works in both Standalone and Domain modes.

Execute following command using the CLI once you connected to the Server. Make sure you provide the correct directory name and other properties below. Add any additional properties required by the connector by duplicating the "connection-definitions" command below. Edit the JNDI name to match the JNDI name you used in VDB.

```
batch
/subsystem=resource-adapters/resource-adapter=file/connection-definitions=fileDS:add(jndi-name=java:/fileDS, class-name=org.teiid.resource.adapter.file.FileManagedConnectionFactory, enabled=true, use-java-context=true)
/subsystem=resource-adapters/resource-adapter=file/connection-definitions=fileDS/config-properties=Parentdirectory:add(value=/home/rareddy/testing/)
/subsystem=resource-adapters/resource-adapter=file/connection-definitions=fileDS/config-properties=AllowParentPaths:add(value=true)
/subsystem=resource-adapters/resource-adapter=file:activate
runbatch
```

To find out all the properties that are supported by this File Connector execute the following command in the CLI.

```
/subsystem=teiid:read-rar-description(rar-name=file)
```

Tip	**Developer’s Tip** - If WildFly 19.1.0 is running in standalone mode, you can also manually edit the "<jboss-install>/standalone/configuration/standalone-teiid.xml" file and add the XML configuration defined in "<jboss-install>/docs/teiid/datasources/file" directory under "resource-adapters" subsystem. Shutdown the server before you edit this file, and restart after the modifications are done.

Ftp/Ftps Data Sources

Ftp/Ftps data sources use a Teiid specific JCA connector that is deployed into WildFly 19.1.0 during installation. There are many ways to create the Ftp/Ftps data source, using CLI, admin-console, etc. The example shown below uses the CLI tool, as this works in both Standalone and Domain modes.

Execute following command using the CLI once you connected to the Server. Make sure you provide the correct directory name and other properties below. Add any additional properties required by the connector by duplicating the "connection-definitions" command below. Edit the JNDI name to match the JNDI name you used in VDB.

```
/subsystem=resource-adapters/resource-adapter=ftp:add(module=org.jboss.teiid.resource-adapter.ftp)
/subsystem=resource-adapters/resource-adapter=ftp/connection-definitions=ftpDS:add(jndi-name=${jndi.name}", cla
ss-name=org.teiid.resource.adapter.ftp.FtpManagedConnectionFactory, enabled=true, use-java-context=true)
/subsystem=resource-adapters/resource-adapter=ftp/connection-definitions=ftpDS/config-properties=ParentDirector
y:add(value="${ftp.parent.dir}")
/subsystem=resource-adapters/resource-adapter=ftp/connection-definitions=ftpDS/config-properties=Host:add(value
="${ftp.parent.host}")
/subsystem=resource-adapters/resource-adapter=ftp/connection-definitions=ftpDS/config-properties=Port:add(value
=${ftp.parent.port}")
/subsystem=resource-adapters/resource-adapter=ftp/connection-definitions=ftpDS/config-properties=Username:add(v
alue=${ftp.parent.username}")
/subsystem=resource-adapters/resource-adapter=ftp/connection-definitions=ftpDS/config-properties=Password:add(v
alue=${ftp.parent.password}")
/subsystem=resource-adapters/resource-adapter=ftp:activate()
```

To find out all the properties that are supported by this Ftp/Ftps Connector execute the following command in the CLI.

```
/subsystem=teiid:read-rar-description(rar-name=ftp)
```

Tip	**Developer’s Tip** - If WildFly 19.1.0 is running in standalone mode, you can also manually edit the "<jboss-install>/standalone/configuration/standalone-teiid.xml" file and add the XML configuration defined in "<jboss-install>/docs/teiid/datasources/ftp" directory under "resource-adapters" subsystem. Shutdown the server before you edit this file, and restart after the modifications are done.

Google Spreadsheet Data Sources

The Google JCA connector is named teiid-connector-google.rar. The examples include a sample google.xml file. The JCA connector has number of config-properties to drive authentication. The JCA connector connects to exactly one spreadsheet with each sheet exposed as a table.

Authentication to your google account may be done using OAuth, which requires a refresh token (outlined below).

Config property	Description
ClientId	client ID for access. If not specified, the Teiid default will be used.
ClientSecret	client secret for access. If not specified, the Teiid default will be used.
RefreshToken	Use guide below to retrieve RefreshToken. Request access to Google Drive and Spreadsheet API.
SpreadsheetName	Name/Title of the Spreadsheet.
SpreadsheetId	ID of Spreadsheet.
ApiVersion	Optional GData API version. Can be v3 or v4. Defaults to v3.
BatchSize	Maximum number of rows that can be fetched at a time. Defaults to 4096.

The v4 api requires the use of SpreadsheetId and specifying ClientId and ClientSecret. Some sheets such as those contained in a team drive will only be visible to the v4 api.

Create Authorization Credentials

For v3 connections it is recommended that you create your own authorization credentials rather than relying on the default Teiid client id and client secret. For v4 connections it is required that you create your own credentials. Creating your own project will give you greater control over monitoring and controlling API access.

You should follow the OAuth2 For Devices Guide prerequisites. You should allow the project access to Google Drive API and the Sheets API.

A condensed form of the rest of the guide "Obtaining OAuth 2.0 access tokens" is covered next as "Getting an OAuth Refresh Token".

Getting an OAuth Refresh Token

With a browser or other client issue the request with the appropriate client ID:

https://accounts.google.com/o/oauth2/auth?scope=https%3A%2F%2Fwww.googleapis.com%2Fauth%2Fdrive+https%3A%2F%2Fwww.googleapis.com%2Fauth%2Fspreadsheets&redirect_uri=urn:ietf:wg:oauth:2.0:oob&response_type=code&client_id=<CLIENT_ID>;

Then copy the authorization code into following POST request and run it in command line:

```
curl \--data-urlencode code=<AUTHORIZATION_CODE> \
--data-urlencode client_id=<CLIENT_ID> \
--data-urlencode client_secret=<CLIENT_SECRET> \
--data-urlencode redirect_uri=urn:ietf:wg:oauth:2.0:oob \
--data-urlencode grant_type=authorization_code https://accounts.google.com/o/oauth2/token
```

The refresh token will be in the response.

To use the Teiid defaults:

Click on https://accounts.google.com/o/oauth2/auth?scope=https%3A%2F%2Fwww.googleapis.com%2Fauth%2Fdrive+https%3A%2F%2Fspreadsheets.google.com%2Ffeeds&redirect_uri=urn:ietf:wg:oauth:2.0:oob&response_type=code&client_id=217138521084.apps.googleusercontent.com

Then copy the authorization code into following POST request and run it in command line:

```
curl \--data-urlencode code=<AUTHORIZATION_CODE> \
--data-urlencode client_id=217138521084.apps.googleusercontent.com \
--data-urlencode client_secret=gXQ6-lOkEjE1lVcz7giB4Poy \
--data-urlencode redirect_uri=urn:ietf:wg:oauth:2.0:oob \
--data-urlencode grant_type=authorization_code https://accounts.google.com/o/oauth2/token
```

Implementation Details

Google Translator is implemented using GData API and the Google Visualization Protocol. v4 connections still rely upon v3 functionality for update/delete as the v4 API does not provide appropriate search functionality.

Infinispan Data Sources

Infinispan data sources use a Teiid specific JCA connector that is deployed into WildFly 19.1.0 during installation. There are many ways to create a Infinispan hotrod based data source, using CLI, admin-console, etc. The example shown below uses the CLI tool, as this works in both Standalone and Domain modes.

Execute the following commands using the CLI once you connected to the Server. Make sure you provide the correct URL and user credentials. Add any additional properties required by the connector by duplicating the "connection-definitions" command below. Edit the JNDI name to match the JNDI name you used in VDB.

```
batch
/subsystem=resource-adapters/resource-adapter=infinispanDS:add(module=org.jboss.teiid.resource-adapter.infinisp
an.hotrod)
/subsystem=resource-adapters/resource-adapter=infinispanDS/connection-definitions=ispnDS:add(jndi-name="java:/i
spnDS", class-name=org.teiid.resource.adapter.infinispan.hotrod.InfinispanManagedConnectionFactory, enabled=true
, use-java-context=true)
/subsystem=resource-adapters/resource-adapter=infinispanDS/connection-definitions=ispnDS/config-properties=Remo
teServerList:add(value="{host}:11222")
/subsystem=resource-adapters/resource-adapter=infinispanDS:activate
run-batch
```

All the properties that are defined on the RAR file are

Property Name	Description	Required	Default
RemoteServerList	A comma separated list of server locations. Each location can contain an optional port, of the format host:port	Yes	n/a
UserName	If remote server is secured, this property is used as username to login	No	n/a
Password	If remote server is secured, this property is used as password to login	No	n/a
SaslMechanism	"EXTERNAL" is when certificate based security at use, all others use username/password.	No	The default mechanism of Hotrod.
AuthenticationRealm	Realm to use for authentication.	No	n/a
AuthenticationServerName	Infinispan server name where the Authentication is handled.	No	n/a
TrustStoreFileName	When "EXTERNAL" SaslMechnism used, use this property to define truststore. Alternatively JAVA system property "javax.net.ssl.trustStore" can also be defined instead.	No	n/a

TrustStorePassword	When "EXTERNAL" SaslMechnism used, use this property to define truststore password. Alternatively JAVA system property "javax.net.ssl.trustStorePassword" can also be defined instead.	No	n/a
KeyStoreFileName	When "EXTERNAL" SaslMechnism used, use this property to define keystore. Alternatively JAVA system property "javax.net.ssl.keyStore" can also be defined instead.	No	n/a
KeyStorePassword	When "EXTERNAL" SaslMechnism used, use this property to define keystore password. Alternatively JAVA system property "javax.net.ssl.keyStorePassword" can also be defined instead.	No	n/a
CacheName	The default cache name.	No	n/a
CacheTemplate	If a cache needs to be created the template name to use.	No	n/a
TransactionMode	The transaction mode expected for cache access. Can be one of: FULL_XA, NON_DURABLE_XA, NON_XA	No	n/a

To find out all the properties that are supported by this Infinispan Connector, execute the following command in the CLI.

```
/subsystem=teiid:read-rar-description(rar-name=infinispan)
```

Tip	**Developer’s Tip** - If WildFly 19.1.0 is running in standalone mode, you can also manually edit the "<jboss-install>/standalone/configuration/standalone-teiid.xml" file and add the XML configuration defined in **"<jboss-install>/docs/teiid/datasources/infinispan"** directory under "resource-adapters" subsystem. Shutdown the server before you edit this file, and restart after the modifications are done.

JDBC Data Sources

The following is an example highlighting configuring an Oracle data source. The process is nearly identical regardless of the database vendor. Typically the JDBC jar and the configuration like connection URL and user credentials change.

There are configuration templates for all the data sources in the "<jboss-install>/docs/teiid/datasources" directory. A complete description how a data source can be added into WildFly is also described here. The below we present two different ways to create a datasource.

Deploying a single JDBC Jar File

First step in configuring the data source is deploying the required JDBC jar file. For example, if you are trying to create a Oracle data source, first you need to deploy the "ojdbc6.jar" file first. Execute following command using the CLI once you connected to the Server.

```
deploy /path/to/ojdbc6.jar
```

Tip	**Developer’s Tip** - If WildFly 19.1.0 is running in standalone mode, you can also manually copy this ’ojdbc6.jar" to the "<jboss-install>/standalone/deployments" directory, to automatically deploy without using the CLI tool.

Creating a module for the Driver

You may also create a module to have more control over the handling of the driver. In cases where the driver is not contained in a single file, this may be preferable to creating a "uber" jar as the dependencies can be managed separately.

Creating a module for a driver is no different than any other container module. You just include the necessary jars as resources in the module and reference other modules as dependencies.

```
<module xmlns="urn:jboss:module:1.0" name="com.mysql">
  <resources>
    <resource-root path="mysql-connector-java-5.1.21.jar"/>
  </resources>
  <dependencies>
    <module name="javax.api"/>
    ...
  </dependencies>
</module>
```

Create Data Source

Now that you have the JDBC driver deployed or the module created, it is time to create a data source using this driver. There are many ways to create the datasource using CLI, admin-console etc. The example shown below uses the CLI tool, as this works in both Standalone and Domain modes.

Execute following command using CLI once you connected to the Server. Make sure you provide the correct URL and user credentials and edit the JNDI name to match the JNDI name you used in VDB.

```
/subsystem=datasources/data-source=oracel-ds:add(jndi-name=java:/OracleDS, driver-name=ojdbc6.jar, connection-u
rl=jdbc:oracle:thin:{host}:1521:orcl,user-name={user}, password={password})
/subsystem=datasources/data-source=oracel-ds:enable
```

The driver-name will match the name of jar or module that you deployed for the driver.

Tip	**Developer’s Tip** - If WildFly 19.1.0 is running in standalone mode, you can also manually edit the "<jboss-install>/standalone/configuration/standalone-teiid.xml" file and add the XML configuration defined in *"<jboss-install>/docs/teiid/datasources/oracle" directory under "datasources" subsystem. Shutdown the server before you edit this file, and restart after the modifications are done.

LDAP Data Sources

LDAP data sources use a Teiid specific JCA connector which is deployed into WildFly 19.1.0 during installation. There are many ways to create the ldap data source, using CLI, admin-console etc. The example shown below uses the CLI tool, as this works in both Standalone and Domain modes.

Execute following command using CLI once you connected to the Server. Make sure you provide the correct URL and user credentials. Add any additional properties required by the connector by duplicating the "connection-definitions" command below. Edit the JNDI name to match the JNDI name you used in VDB.

```
batch
/subsystem=resource-adapters/resource-adapter=ldap/connection-
definitions=ldapDS:add(jndi-name=java:/ldapDS, class-
name=org.teiid.resource.adapter.ldap.LDAPManagedConnectionFactory, enabled=true,
use-java-context=true)
/subsystem=resource-adapters/resource-adapter=ldap/connection-
definitions=ldapDS/config-properties=LdapUrl:add(value=ldap://ldapServer:389)
/subsystem=resource-adapters/resource-adapter=ldap/connection-
definitions=ldapDS/config-properties=LdapAdminUserDN:add(value=
{cn=???,ou=???,dc=???})
/subsystem=resource-adapters/resource-adapter=ldap/connection-
definitions=ldapDS/config-properties=LdapAdminUserPassword:add(value={pass})
/subsystem=resource-adapters/resource-adapter=ldap/connection-
definitions=ldapDS/config-properties=LdapTxnTimeoutInMillis:add(value=-1)
/subsystem=resource-adapters/resource-adapter=ldap:activate
runbatch
```

To find out all the properties that are supported by this LDAP Connector execute the following command in the CLI.

```
/subsystem=teiid:read-rar-description(rar-name=ldap)
```

Tip	**Developer’s Tip** - If WildFly 19.1.0 is running in standalone mode, you can also manually edit the "<jboss-install>/standalone/configuration/standalone-teiid.xml" file and add the XML configuration defined in *"<jboss-install>/docs/teiid/datasources/ldap" directory under "resource-adapters" subsystem. Shutdown the server before you edit this file, and restart after the modifications are done.

Note	To use an anonymous bind, set the LdapAuthType to none. When performing an anonymous bind the values for the admin user and password will be ignored.

Tip	If you experience stale connections in the pool, you should enable either the validate-on-match or the background-validation pool settings.

MongoDB Data Sources

MongoDB data sources use a Teiid specific JCA connector that is deployed into WildFly 19.1.0 during installation. There are many ways to create a MongoDB data source, using CLI, admin-console, etc. The example shown below uses the CLI tool, as this works in both Standalone and Domain modes.

Execute the following command using the CLI once you connected to the Server. Make sure you provide the correct URL and user credentials. Add any additional properties required by the connector by duplicating the "connection-definitions" command below. Edit the JNDI name to match the JNDI name you used in VDB.

```
batch
/subsystem=resource-adapters/resource-adapter=mongodb/connection-definitions=mongodbDS:add(jndi-name="java:/mon
goDS", class-name=org.teiid.resource.adapter.mongodb.MongoDBManagedConnectionFactory, enabled=true, use-java-co
ntext=true)
/subsystem=resource-adapters/resource-adapter=mongodb/connection-definitions=mongodbDS/config-properties=Remote
ServerList:add(value="{host}:27017")
/subsystem=resource-adapters/resource-adapter=mongodb/connection-definitions=mongodbDS/config-properties=Databa
se:add(value="{db-name}")
/subsystem=resource-adapters/resource-adapter=mongodb:activate
runbatch
```

All the properties that are defined on the RAR file are

Property Name	Description	Required	Default
RemoteServerList	A comma separated list of server locations. Each location can contain an optional port, of the format host:port. The property may also contain a full standard (mongodb://) or seedlist (mongodb+srv://) connection URI string. If a full connection string is used, then none of the other configuration properties will be used nor are required. However the database should be specified in the URI.	true	
Username	Connection User’s Name	false	none
Password	Connection User’s password	false	none
Database	MongoDB database name - required if not suing a full connection URI in the RemoteServerList	false	none
SecurityType	MongoDB Type of Authentication to be used. Allowed values are "None","SCRAM_SHA_1", "SCRAM_SHA_256","MONGODB_CR", "Kerberos","X509". If you are using MongoDB version lessthan 3.0, MongoDB by default uses "MONGODB_CR", thus this value need to be set accordingly or set to None.	false	SCRAM_SHA_1
AuthDatabase	MongoDB Database Name for user authentication in case when SecurityType 'MONGODB-CR' is used. This is an optional value.	false	none

Ssl	Use SSL Connections	false	none

To find out all the properties that are supported by this MongoDB Connector execute the following command in the CLI.

```
/subsystem=teiid:read-rar-description(rar-name=mongodb)
```

Tip	**Developer’s Tip** - If WildFly 19.1.0 is running in standalone mode, you can also manually edit the "<jboss-install>/standalone/configuration/standalone-teiid.xml" file and add the XML configuration defined in **"<jboss-install>/docs/teiid/datasources/mongodb"** directory under "resource-adapters" subsystem. Shutdown the server before you edit this file, and restart after the modifications are done.

Transaction support

With Teiid 12.1 running against a MongoDB server 4+ in a replica set, you may optionally use LocalTransaction transaction support. Doing so through the CLI requires the creation of a new resource adapter.

```
batch
/subsystem=resource-adapters/resource-adapter=mongodbLocal:add(transaction-support=LocalTransaction, module=org
.jboss.teiid.resource-adapter.mongodb:main
/subsystem=resource-adapters/resource-adapter=mongodbLocal/connection-definitions=mongodbLocal:add(jndi-name="j
ava:/mongoDS", class-name=org.teiid.resource.adapter.mongodb.MongoDBManagedConnectionFactory, enabled=true, use
-java-context=true)
...
runbatch
```

Phoenix Data Sources

The following is a example for setting up Phoenix Data Sources, which is precondition for Apache Phoenix Translator. In addition to the Data Sources set up, this article also cover mapping Phoenix table to an existing HBase table and creating a new Phoenix table.

There are configuration templates for Phoenix data sources in the "<jboss-install>/docs/teiid/datasources" directory. A complete description how a data source can be added into WildFly is also described here.

Configuring a Phoenix data source in WildFly

Configuring a Phoenix data source is nearly identical to configuring JDBC Data Sources. The first step is deploying the Phoenix driver jar. Using below CLI command to deploy Phoenix driver:

```
module add --name=org.apache.phoenix  --resources=/path/to/phoenix-[version]-client.jar  --dependencies=javax.a
pi,sun.jdk,org.apache.log4j,javax.transaction.api
/subsystem=datasources/jdbc-driver=phoenix:add(driver-name=phoenix,driver-module-name=org.apache.phoenix,driver-
class-name=org.apache.phoenix.jdbc.PhoenixDriver)
```

The Driver jar can be download from phoenix document.

The second steps is creating the Data Source base on above deployed driver, which is also like creating JDBC Data Source. Using below CLI command to create Data Source:

```
/subsystem=datasources/data-source=phoenixDS:add(jndi-name=java:/phoenixDS,  driver-name=phoenix, connection-ur
l=jdbc:phoenix:{zookeeper quorum server}, enabled=true, use-java-context=true, user-name={user}, password={pass
word})
/subsystem=datasources/data-source=phoenixDS/connection-properties=phoenix.connection.autoCommit:add(value=true)
```

Please make sure the URL, Driver, and other properties are configured correctly:

- jndi-name - The JNDI name need to match the JNDI name you used in VDB
- driver-name - The Driver name need to match the driver you deployed in above steps
- connection-url - The URL need to match the HBase zookeeper quorum server, the format like *jdbc:phoenix [:<zookeeper quorum> [:<port number>] [:<root node>]], 'jdbc:phoenix:127.0.0.1:2181'* is a example
- user-name/password - The user credentials for Phoenix Connection

The Phoenix Connection AutoCommit default is false. Set *phoenix.connection.autoCommit* to true if you will be executing INSERT/UPDATE/DELETE statements against Phoenix.

Mapping Phoenix table to an existing HBase table

Mapping Phoenix table to an existing HBase table has 2 steps. The first step is installing phoenix-[version]-server.jar to the classpath of every HBase region server. An easy way to do this is to copy it into the HBase lib - for more details please refer to the phoenix documentation.

The second step is executing the DDL to map a Phoenix table to an existing HBase table. The DDL can either be executed via Phoenix Command Line, or executed by JDBC.

The Following is a example for mapping an existing HBase Customer with the following structure:

Row Key	customer		sales	
ROW_ID	name	city	product	amount
101	John White	Los Angeles, CA	Chairs	$400.00
102	Jane Brown	Atlanta, GA	Lamps	$200.00
103	Bill Green	Pittsburgh, PA	Desk	$500.00
104	Jack Black	St. Louis, MO	Bed	$1,600.00

As depicted above, the HBase *Customer* table have 2 column families, *customer* and *sales*, and each has 2 column qualifiers, *name*, *city*, *product* and *amount* respectively. We can map this Table to Phoenix via DDL:

```
CREATE TABLE IF NOT EXISTS "Customer"("ROW_ID" VARCHAR PRIMARY KEY, "customer"."city" VARCHAR, "customer"."name" VARCHAR, "sales"."amount" VARCHAR, "sales"."product" VARCHAR)
```

For more about mapping Phoenix table to an existing HBase table please refer to the phoenix documentation.

Creating a new Phoenix table

Creating a new Phoenix table is just like mapping to an existing HBase table. Phoenix will create any metadata (table, column families) that do not exist. Similar to the above example the DDL to create the Phoenix/HBase Customer table would be:

```
CREATE TABLE IF NOT EXISTS "Customer"("ROW_ID" VARCHAR PRIMARY KEY, "customer"."city" VARCHAR, "customer"."name" VARCHAR, "sales"."amount" VARCHAR, "sales"."product" VARCHAR)
```

Defining Foreign Table in VDB

Finally, we need define a Foreign Table in VDB that map to Phoenix table, the following principles should be considered in defining Foreign Table:

- nameinsource option in Table used to match Phoenix table name
- nameinsource option in Column used to match HBase Table’s Columns
- create a primary key is recommended, the primary key column should match Phoenix table’s primary key/HBase row id.

With "*Mapping Phoenix table to an existing HBase table*" section’s `Customer' table, below is a example:

```
CREATE FOREIGN TABLE Customer (
    PK string OPTIONS (nameinsource 'ROW_ID'),
    city string OPTIONS (nameinsource '"city"'),
    name string OPTIONS (nameinsource '"name"'),
    amount string OPTIONS (nameinsource '"amount"'),
    product string OPTIONS (nameinsource '"product"'),
    CONSTRAINT PK0 PRIMARY KEY(PK)
) OPTIONS(nameinsource '"Customer"', "UPDATABLE" 'TRUE');
```

Note	"Constraint violation. X may not be null" exception may thrown if updating a table without defining a primary key.

OSISoft PI Data Sources

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 queries.

Salesforce Data Sources

Salesforce data sources use a Teiid specific JCA connector that is deployed into WildFly 19.1.0 during installation. There are three versions of the salesforce resource adapter - salesforce, which currently provides connectivity to the 34.0 Salesforce API, salesforce-34, which provides connectivity to the 34.0 Salesforce API, and salesforce-41 which actually provides access to 37.0 through at least 45.0. The version 22.0 support has been removed.

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.

There are many ways to create the salesforce data source, using CLI,AdminAPI, admin-console etc. The example shown below uses the CLI tool, as this works in both Standalone and Domain modes.

Execute following command using the CLI once you connected to the Server. Make sure you provide the correct URL and user credentials. Add any additional properties required by the connector by duplicating the "connection-definitions" command below. Edit the JNDI name to match the JNDI name you used in VDB.

```
batch
/subsystem=resource-adapters/resource-adapter=salesforce/connection-definitions=sfDS:add(jndi-name=java:/sfDS,
class-name=org.teiid.resource.adapter.salesforce.SalesForceManagedConnectionFactory, enabled=true, use-java-con
text=true)
/subsystem=resource-adapters/resource-adapter=salesforce/connection-definitions=sfDS/config-properties=URL:add(
value=https://login.salesforce.com/services/Soap/u/34.0)
/subsystem=resource-adapters/resource-adapter=salesforce/connection-definitions=sfDS/config-properties=username
:add(value={user})
/subsystem=resource-adapters/resource-adapter=salesforce/connection-definitions=sfDS/config-properties=password
:add(value={password})
/subsystem=resource-adapters/resource-adapter=salesforce:activate
runbatch
```

The salesforce-xx connection definition configuration is similar to the above. The resource adapter name would instead be salesforce-xx, and the url would point to a later version. It is recommended to set the url explicitly. If you use just the salesforce resource adapter without setting the url, then later versions of Teiid can use a different default once the resource adapter moves to a different version. The -34 resource adapter defaults to https://login.salesforce.com/services/Soap/u/34.0, and the -41 resource adapter defaults to https://login.salesforce.com/services/Soap/u/40.0

Note	that if you access a newer Salesforce API version than the resource adapter supports, you will receive low level metadata parsing exceptions - you can either access an older API or log an issue to have updated support.

To find out all the properties that are supported by this Salesforce Connector execute the following command in the CLI.

```
/subsystem=teiid:read-rar-description(rar-name=salesforce)
```

Tip	**Developer’s Tip** - If WildFly 19.1.0 is running in standalone mode, you can also manually edit the "<jboss-install>/standalone/configuration/standalone-teiid.xml" file and add the XML configuration defined in *"<jboss-install>/docs/teiid/datasources/salesforce" directory under "resource-adapters" subsystem. Shutdown the server before you edit this file, and restart after the modifications are done.

Mutual Authentication

If you need to connect to Salesforce using Mutual Authentication, follow the directions to setup Salesforce at https://help.salesforce.com/apex/HTViewHelpDoc?id=security_keys_uploading_mutual_auth_cert.htm&language=en_US then configure the below CXF configuration file on the resource-adapter by adding following property to above cli script

```
/subsystem=resource-adapters/resource-adapter=salesforce/connection-definitions=sfDS/config-properties=ConfigFile:add(value=${jboss.server.config.dir}/cxf-https.xml)
```

cxf-https.xml

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:sec="http://cxf.apache.org/configuration/security"
    xmlns:http-conf="http://cxf.apache.org/transports/http/configuration"
    xmlns:jaxws="http://java.sun.com/xml/ns/jaxws"
    xsi:schemaLocation="http://cxf.apache.org/transports/http/configuration http://cxf.apache.org/schemas/configuration/http-conf.xsd http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.0.xsd http://cxf.apache.org/configuration/security http://cxf.apache.org/schemas/configuration/security.xsd">

    <http-conf:conduit name="*.http-conduit">
        <http-conf:client ConnectionTimeout="120000" ReceiveTimeout="240000"/>
        <http-conf:tlsClientParameters secureSocketProtocol="SSL">
          <sec:trustManagers>
            <sec:keyStore type="JKS" password="changeit" file="/path/to/truststore.jks"/>
          </sec:trustManagers>
        </http-conf:tlsClientParameters>
    </http-conf:conduit>
</beans>
```

more information about CXF configuration file can be found at http://cxf.apache.org/docs/client-http-transport-including-ssl-support.html#ClientHTTPTransport(includingSSLsupport)-ConfiguringSSLSupport

OAuth Security with "Refresh Token"

The below layout the directions to use Refresh Token based OAuth Authentication with Salesforce.

1) create connected app (may need to setup custom domain) 2) add profile and/or permissions set to the connected app 3) grab the "callback url" (one need to set as https://localhost:443/_callback" 4) Run through the teiid-oauth-util.sh in "<eap>/bin" directory, use client_id, client_pass, and call back from connected app 5) use "https://login.salesforce.com/services/oauth2/authorize" authorize link 6) use "https://login.salesforce.com/services/oauth2/token" for access token url 7) the you get a refresh token from it 8) create a security-domain by executing CLI

```
/subsystem=security/security-domain=oauth2-security:add(cache-type=default)
/subsystem=security/security-domain=oauth2-security/authentication=classic:add
/subsystem=security/security-domain=oauth2-security/authentication=classic/login-module=Kerberos:add(code=org.teiid.jboss.oauth.OAuth20LoginModule, flag=required, module=org.jboss.teiid.security,
    module-options=[client-id=xxxx, client-secret=xxxx, refresh-token=xxxx,
    access-token-uri=https://login.salesforce.com/services/oauth2/token])
reload
```

this will generate following XML in the standalone.xml or domain.xml (this can also be directly added to the standalone.xml or domain.xml files instead of executing the CLI)

standalone.xml

```
<security-domain name="oauth2-security">
```

```
    <authentication>
        <login-module code="org.teiid.jboss.oauth.OAuth20LoginModule" flag="required" module="org.jboss.teiid.s
ecurity">
            <module-option name="client-id" value="xxxx"/>
            <module-option name="client-secret" value="xxxx"/>
            <module-option name="refresh-token" value="xxxx"/>
            <module-option name="access-token-uri" value="https://login.salesforce.com/services/oauth2/token"/>
        </login-module>
    </authentication>
</security-domain>
```

9) Then to use the above security domain in the sales force data source configuration, add "<security-domain>oauth2-security</security-domain>"

OAuth Security with "JWT Token" based Steps

The below layout the directions to use JWT token based OAuth Authentication with Salesforce.

1) Create a Self-Signed certificate locally or on Sales Force. (user → setup → security-controls → Certificate and Key Management) 2) Download the certificate and also put in keystore and download keystore. Keystore is needed for Teiid, certificate for the salesforce setup 3) Create connected app and select OAuth, and select all the scopes (some posts say refresh-token offline is must) 4) create a profile and/or permission set assign to the connected app. I believe before you can create a connected app you need to set up custom domain 5) When you creating connected app make sure you add the certificate in "Digital Certificate" 6) Now in Teiid create security-domain by executing CLI

```
/subsystem=security/security-domain=oauth2-jwt-security:add(cache-type=default)
/subsystem=security/security-domain=oauth2-jwt-security/authentication=classic:add
/subsystem=security/security-domain=oauth2-jwt-
security/authentication=classic/login-
module=oauth:add(code=org.teiid.jboss.oauth.OAuth20LoginModule, flag=required,
module=org.jboss.teiid.security,
   module-options=[client-id=xxxx, client-secret=xxxx, access-token-
uri=https://login.salesforce.com/services/oauth2/token, jwt-
audience=https://login.salesforce.com, jwt-subject=your@sf-login.com,
    keystore-type=JKS, keystore-password=changeme, keystore-
url=${jboss.server.config.dir}/salesforce.jks, certificate-alias=teiidtest,
signature-algorithm-name=SHA256withRSA])
reload
```

this will generate following XML in the standalone.xml or domain.xml (this can also be directly added to the standalone.xml or domain.xml files instead of executing the CLI)

standalone.xml

```
<security-domain name="oauth2-jwt-security">
    <authentication>
        <login-module code="org.teiid.jboss.oauth.JWTBearerTokenLoginModule" flag="required" module="org.jboss.
teiid.security">
            <module-option name="client-id" value="xxxxx"/>
            <module-option name="client-secret" value="xxxx"/>
            <module-option name="access-token-uri" value="https://login.salesforce.com/services/oauth2/token"/>
            <module-option name="jwt-audience" value="https://login.salesforce.com"/>
            <module-option name="jwt-subject" value="your@sf-login.com"/>

            <module-option name="keystore-type" value="JKS"/>
            <module-option name="keystore-password" value="changeme"/>
            <module-option name="keystore-url" value="${jboss.server.config.dir}/salesforce.jks"/>
```

```
            <module-option name="certificate-alias" value="teiidtest"/>
            <module-option name="signature-algorithm-name" value="SHA256withRSA"/>
        </login-module>
    </authentication>
</security-domain>
```

7) Then to use the above security domain in the sales force data source configuration, add "<security-domain>oauth2-jwt-security</security-domain>"

More helpful links

https://developer.salesforce.com/blogs/developer-relations/2011/03/oauth-and-the-soap-api.html
https://help.salesforce.com/apex/HTViewHelpDoc?id=remoteaccess_oauth_jwt_flow.htm&language=en_US#create_token
http://salesforce.stackexchange.com/questions/31904/how-and-when-does-a-salesforce-saml-oauth2-user-give-permission-to-use-a-conne http://salesforce.stackexchange.com/questions/30596/oauth-2-0-jwt-bearer-token-flow
http://salesforce.stackexchange.com/questions/88396/invalid-assertion-error-in-jwt-bearer-token-flow

Logging

Logging, when enabled, will be performed at an INFO level to the org.apache.cxf.interceptor context.

Per Resource Adapder

The CXF config property may also be used to control the logging of requests and responses.

Example logging data source

```
<resource-adapter id="salesforce-ds">
    <module slot="main" id="org.jboss.teiid.resource-adapter.salesforce-34"/>
    <transaction-support>NoTransaction</transaction-support>
    <connection-definitions>
        <connection-definition class-name="org.teiid.resource.adapter.salesforce.SalesForceManagedConnectionFac
tory" jndi-name="java:/salesforce_bulk_api" enabled="true" use-java-context="true" pool-name="salesforce-ds">
            <config-property name="password">
                token
            </config-property>
            <config-property name="URL">
                https://login.salesforce.com/services/Soap/u/34.0
            </config-property>
            <config-property name="username">
                name
            </config-property>
            <config-property name="ConfigFile">
                /path/to/cxf.xml
            </config-property>
        </connection-definition>
    </connection-definitions>
</resource-adapter>
```

Corresponding cxf.xml

Example logging data source

```
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:cxf="http://cxf.apache.org/core"
      xsi:schemaLocation="http://cxf.apache.org/core http://cxf.apache.org/schemas/core.xsd http://www.springfr
amework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">

    <bean id="loggingFeature" class="org.apache.cxf.feature.LoggingFeature"/>
    <cxf:bus>
        <cxf:features>
```

```
            <ref bean="loggingFeature"/>
        </cxf:features>
    </cxf:bus>
</beans>
```

All CXF Usage

With the WildFly distribution of CXF a system property can be used to enable CXF logging across all usage in the application server - see the WildFly docs.

Example System Property

```
<system-properties>
  <property name="org.apache.cxf.logging.enabled" value="true"/>
</system-properties>
```

Solr Data Sources

Solr data sources use a Teiid specific JCA connector that is deployed into WildFly 19.1.0 during installation. There are many ways to create a Solr data source, using CLI, admin-console, etc. The example shown below uses the CLI tool, as this works in both Standalone and Domain modes.

Execute the following command using the CLI once you connected to the Server. Make sure you provide the correct URL and user credentials. Add any additional properties required by the connector by duplicating the "connection-definitions" command below. Edit the JNDI name to match the JNDI name you used in VDB.

```
batch
/subsystem=resource-adapters/resource-adapter=solr/connection-definitions=solrDS:add(jndi-name=java:/solrDS, class-name=org.teiid.resource.adapter.solr.SolrManagedConnectionFactory, enabled=true, use-java-context=true)
/subsystem=resource-adapters/resource-adapter=solr/connection-definitions=solrDS/config-properties=url:add(value=http://localhost:8983/solr/)
/subsystem=resource-adapters/resource-adapter=solr/connection-definitions=solrDS/config-properties=CoreName:add(value=collection1)
/subsystem=resource-adapters/resource-adapter=solr:activate
runbatch
```

To find out all the properties that are supported by this Solr Connector execute the following command in the CLI.

```
/subsystem=teiid:read-rar-description(rar-name=solr)
```

Tip	**Developer’s Tip** - If WildFly 19.1.0 is running in standalone mode, you can also manually edit the "<jboss-install>/standalone/configuration/standalone-teiid.xml" file and add the XML configuration defined in **"<jboss-install>/docs/teiid/datasources/solr"** directory under "resource-adapters" subsystem. Shutdown the server before you edit this file, and restart after the modifications are done.

Web Service Data Sources

Web service data sources use a Teiid specific JCA connector that is deployed into WildFly 19.1.0 during installation. There are many ways to create the file data source, using CLI, admin-console etc. The example shown below uses the CLI tool, as this works in both Standalone and Domain modes.

Execute following command using the CLI once you connected to the Server. Make sure you provide the correct endpoint and other properties below. Add any additional properties required by the connector by duplicating the "connection-definitions" command below. Edit the JNDI name to match the JNDI name you used in VDB.

```
batch
/subsystem=resource-adapters/resource-adapter=webservice/connection-definitions=wsDS:add(jndi-name=java:/wsDS,
class-name=org.teiid.resource.adapter.ws.WSManagedConnectionFactory, enabled=true, use-java-context=true)
/subsystem=resource-adapters/resource-adapter=webservice/connection-definitions=wsDS/config-properties=EndPoint
:add(value={end_point})
/subsystem=resource-adapters/resource-adapter=webservice:activate
runbatch
```

To find out all the properties that are supported by this Web Service Connector execute the following command in the CLI.

```
/subsystem=teiid:read-rar-description(rar-name=webservice)
```

The Web Service Data Source supports specifying a WSDL using the Wsdl property. If the Wsdl property is set, then the ServiceName, EndPointName, and NamespaceUri properties should also be set. The Wsdl property may be a URL or file location or the WSDL to use.

Tip	**Developer’s Tip** - If WildFly 19.1.0 is running in standalone mode, you can also manually edit the "<jboss-install>/standalone/configuration/standalone-teiid.xml" file and add the XML configuration defined in *"<jboss-install>/docs/teiid/datasources/web-service" directory under "resource-adapters" subsystem. Shutdown the server before you edit this file, and restart after the modifications are done.

All available configuration properties of web resource-adapter

Property Name	applies to	Required	Default Value	Description
EndPoint	HTTP & SOAP	false	n/a	URL for HTTP, Service Endpoint for SOAP. Not required if using HTTP invoke procedures that specify absolute URLs. Will be used as the base URL if an invoke procedure uses a relative URL.
SecurityType	HTTP & SOAP	false	none	Type of Authentication to used with the web service. Allowed values ["None","HTTPBasic","WSSecurity", "Kerberos", "OAuth"]
AuthUserName	HTTP & SOAP	false	n/a	Name value for authentication, used in HTTPBasic and WsSecurity
AuthPassword	HTTP & SOAP	false	n/a	Password value for authentication, used in HTTPBasic and WsSecurity
ConfigFile	HTTP &	false	n/a	CXF client configuration File or URL

ConfigFile	HTTP & SOAP	false	n/a	CXF client configuration File or URL
EndPointName	HTTP & SOAP	false	teiid	Local part of the end point QName to use with this connection, needs to match one defined in cxf file
ServiceName	SOAP	false	n/a	Local part of the service QName to use with this connection
NamespaceUri	SOAP	false	http://teiid.org	Namespace URI of the service QName to use with this connection
RequestTimeout	HTTP & SOAP	false	n/a	Timeout for request
ConnectTimeout	HTTP & SOAP	false	n/a	Timeout for connection
Wsdl	SOAP	false	n/a	WSDL file or URL for the web service

CXF Configuration

Each web service data source may choose a particular CXF config file and port configuration. The `ConfigFile` config property specifies the Spring XML configuration file for the CXF Bus and port configuration to be used by connections. If no config file is specified then the system default configuration will be used.

Only 1 port configuration can be used by this data source. You may explicitly set the local name of the port QName to use via the `ConfigName` property. The namespace URI for the QName in your config file should match your WSDL/namespace setting on the data source or use the default of http://teiid.org. See the CXF Documentation and the sections below on Security, Logging, etc. for examples of using the CXF configuration file.

Sample Spring XML Configuration To Set Timeouts

```
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:http-conf="http://cxf.apache.org/transports/http/configuration"
        xsi:schemaLocation="http://cxf.apache.org/transports/http/configuration
        http://cxf.apache.org/schemas/configuration/http-conf.xsd
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">

    <http-conf:conduit name="{http://teiid.org}configName.http-conduit">
        <http-conf:client ConnectionTimeout="120000" ReceiveTimeout="240000"/>
    </http-conf:conduit>
</beans>
```

In the conduit name {http://teiid.org[http://teiid.org]}configName.http-conduit, the namespace, {http://teiid.org[http://teiid.org]}, may be set via the namespace datasource property. Typically that will only need done when also supplying the wsdl setting. The local name is followed by .http-conduit. It will be based upon the configName setting, with a default value of teiid.

See the CXF documentation for all possible configuration options.

Note	It is not required to use the Spring configuration to set just timeouts. The ConnectionTimeout and ReceiveTimeout can be set via the resource adapter connectTimeout and requestTimeout properties respectively.

Security

To enable the use of WS-Security, the `SecurityType` should be set to WSSecurity. At this time Teiid does not expect a WSDL to describe the service being used. Thus a Spring XML configuration file is not only required, it must instead contain all of the relevant policy configuration. And just as with the general configuration, each data source is limited to specifying only a single port configuration to use.

```
batch
/subsystem=resource-adapters/resource-adapter=webservice/connection-definitions=wsDS:add(jndi-name=java:/wsDS,
class-name=org.teiid.resource.adapter.ws.WSManagedConnectionFactory, enabled=true, use-java-context=true)
/subsystem=resource-adapters/resource-adapter=webservice/connection-definitions=wsDS/config-properties=ConfigFi
le:add(value=${jboss.server.home.dir}/standalone/configuration/xxx-jbossws-cxf.xml)
/subsystem=resource-adapters/resource-adapter=webservice/connection-definitions=wsDS/config-properties=ConfigNa
me:add(value=port_x)
/subsystem=resource-adapters/resource-adapter=webservice/connection-definitions=wsDS/config-properties=Security
Type:add(value=WSSecurity)
/subsystem=resource-adapters/resource-adapter=webservice:activate
runbatch
```

The corresponding xxx-jbossws-cxf.xml file that adds a timestamp to the SOAP header

Example WS-Security enabled data source

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:jaxws="http://cxf.apache.org/jaxws"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans/spring-beans.xsd
          http://cxf.apache.org/jaxws
          http://cxf.apache.org/schemas/jaxws.xsd">

    <jaxws:client name="{http://teiid.org}port_x"
        createdFromAPI="true">
        <jaxws:outInterceptors>
            <bean/>
            <ref bean="Timestamp_Request"/>
        </jaxws:outInterceptors>
    </jaxws:client>

    <bean

        id="Timestamp_Request">
        <constructor-arg>
            <map>
                <entry key="action" value="Timestamp"/>
            <map>
        </constructor-arg>
    </bean>

</beans>
```

Note that the client port configuration is matched to the data source instance by the QName {http://teiid.org}port_x, where the namespace will match your namespace setting or the default of http://teiid.org. The configuration may contain other port configurations with different local names.

For more information on configuring CXF interceptors, please consult the CXF documentation

Kerberos

WS-Security Kerberos is only supported when the WSDL property is defined in resource-adapter connection configuration and only when WSDL Based Procedures are used. WSDL file must contain WS-Policy section, then WS-Policy section is correctly interpreted and enforced on the endpoint. The sample CXF configuration will look like

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:http="http://cxf.apache.org/transports/http/configuration"
    xmlns:jaxws="http://cxf.apache.org/jaxws"
    xmlns:cxf="http://cxf.apache.org/core"
    xmlns:p="http://cxf.apache.org/policy"
    xmlns:sec="http://cxf.apache.org/configuration/security"
    xsi:schemaLocation="http://www.springframework.org/schema/beans           http://www.springframework.org/sc
hema/beans/spring-beans.xsd           http://cxf.apache.org/jaxws                           http://cxf.apache.o
rg/schemas/jaxws.xsd           http://cxf.apache.org/transports/http/configuration   http://cxf.apache.org/sche
mas/configuration/http-conf.xsd           http://cxf.apache.org/configuration/security          http://cxf.apac
he.org/schemas/configuration/security.xsd           http://cxf.apache.org/core http://cxf.apache.org/schemas/co
re.xsd           http://cxf.apache.org/policy http://cxf.apache.org/schemas/policy.xsd">
    <bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer"/>
    <cxf:bus>
        <cxf:features>
            <p:policies/>
            <cxf:logging/>
        </cxf:features>
    </cxf:bus>

    <jaxws:client name="{http://webservices.samples.jboss.org/}HelloWorldPort" createdFromAPI="true">
        <jaxws:properties>
            <entry key="ws-security.kerberos.client">
                <bean class="org.apache.cxf.ws.security.kerberos.KerberosClient">
                    <constructor-arg ref="cxf"/>
                    <property name="contextName" value="alice"/>
                    <property name="serviceName" value="bob@service.example.com"/>
                </bean>
            </entry>
        </jaxws:properties>
    </jaxws:client>
</beans>
```

and you would need to configure the security-domain in the standalone-teiid.xml file under the 'security' subsystem as

```
<security-domain name="alice" cache-type="default">
    <authentication>
        <login-module code="Kerberos" flag="required">
            <module-option name="storeKey" value="true"/>
            <module-option name="useKeyTab" value="true"/>
            <module-option name="keyTab" value="/home/alice/alice.keytab"/>
            <module-option name="principal" value="alice@EXAMPLE.COM"/>
            <module-option name="doNotPrompt" value="true"/>
            <module-option name="debug" value="true"/>
            <module-option name="refreshKrb5Config" value="true"/>
        </login-module>
    </authentication>
</security-domain>
```

for complete list of kerberos properties please refer to this testcase

Logging

Logging, when enabled, will be performed at an INFO level to the org.apache.cxf.interceptor context.

SOAP

The CXF config property may also be used to control the logging of requests and responses for specific or all ports.

Example logging data source

```
batch
/subsystem=resource-adapters/resource-adapter=webservice/connection-
definitions=wsDS:add(jndi-name=java:/wsDS, class-
name=org.teiid.resource.adapter.ws.WSManagedConnectionFactory, enabled=true, use-
java-context=true)
/subsystem=resource-adapters/resource-adapter=webservice/connection-
definitions=wsDS/config-
properties=ConfigFile:add(value=${jboss.server.home.dir}/standalone/configuration/x
xx-jbossws-cxf.xml)
/subsystem=resource-adapters/resource-adapter=webservice/connection-
definitions=wsDS/config-properties=ConfigName:add(value=port_x)
/subsystem=resource-adapters/resource-adapter=webservice:activate
runbatch
```

Corresponding xxx-jbossws-cxf.xml

Example logging data source

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:jaxws="http://cxf.apache.org/jaxws"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans/spring-beans.xsd
          http://cxf.apache.org/jaxws
          http://cxf.apache.org/schemas/jaxws.xsd">

    <jaxws:client name="{http://teiid.org}port_y"
        createdFromAPI="true">
        <jaxws:features>
            <bean class="org.apache.cxf.feature.LoggingFeature"/>
        </jaxws:features>
    </jaxws:client>

</beans>
```

All CXF Usage

With the WildFly distribution of CXF a system property can be used to enable CXF logging across all usage in the application server (including salesforce) - see the WildFly docs.

Example System Property

```
<system-properties>
  <property name="org.apache.cxf.logging.enabled" value="true"/>
</system-properties>
```

Transport Settings

The CXF config property may also be used to control low level aspects of the HTTP transport. See the CXF documentation for all possible options.

Example Disabling Hostname Verification

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:http-conf="http://cxf.apache.org/transports/http/configuration"
       xsi:schemaLocation="http://cxf.apache.org/transports/http/configuration
                           http://cxf.apache.org/schemas/configuration/http-conf.xsd
                           http://www.springframework.org/schema/beans
                             http://www.springframework.org/schema/beans/spring-beans.xsd">

  <http-conf:conduit name="{http://teiid.org}port_z.http-conduit">
     <!-- WARNING ! disableCNcheck=true should NOT be used in production -->
     <http-conf:tlsClientParameters disableCNcheck="true" />

  </http-conf:conduit>
</beans>
```

Configuring SSL Support (Https)

For using the HTTPS, you can configure CXF file as below

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:sec="http://cxf.apache.org/configuration/security"
    xmlns:http-conf="http://cxf.apache.org/transports/http/configuration"
    xmlns:jaxws="http://java.sun.com/xml/ns/jaxws"
    xsi:schemaLocation="http://cxf.apache.org/transports/http/configuration http://cxf.apache.org/schemas/confi
guration/http-conf.xsd http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/
spring-beans-2.0.xsd http://cxf.apache.org/configuration/security http://cxf.apache.org/schemas/configuration/s
ecurity.xsd">

    <http-conf:conduit name="*.http-conduit">
        <http-conf:client ConnectionTimeout="120000" ReceiveTimeout="240000"/>
        <http-conf:tlsClientParameters secureSocketProtocol="SSL">
          <sec:trustManagers>
            <sec:keyStore type="JKS" password="changeit" file="/path/to/truststore.jks"/>
          </sec:trustManagers>
        </http-conf:tlsClientParameters>
    </http-conf:conduit>
</beans>
```

for all the http-conduit based configuration see http://cxf.apache.org/docs/client-http-transport-including-ssl-support.html. You can also configure for HTTPBasic, kerberos, etc.

Kerberos with REST based Services

Note	**"Kerberos in ws-security with SOAP services" -**

Check out the cxf configuration to allow Kerberos in SOAP web services at http://cxf.apache.org/docs/security.html

The kerberos support is based SPNEGO as described in http://cxf.apache.org/docs/client-http-transport-including-ssl-support.html#ClientHTTPTransport%28includingSSLsupport%29-SpnegoAuthentication%28Kerberos%29. There two types of kerberos support

Negotiation

With this configuration, REST service is configured with Kerberos JAAS domain, to negotiate a token, then use it access the web service. For this first create a security domain in standalone.xml file as below

```
<security-domain name="MY_REALM" cache-type="default">
     <authentication>
         <login-module code="Kerberos" flag="required">
             <module-option name="storeKey" value="true"/>

             <module-option name="useKeyTab" value="true"/>
             <module-option name="keyTab" value="/home/username/service.keytab"/>
             <module-option name="principal" value="host/testserver@MY_REALM"/>

             <module-option name="doNotPrompt" value="true"/>
             <module-option name="debug" value="false"/>
             <module-option name="addGSSCredential" value="true"/>
         </login-module>
     </authentication>
</security-domain>
```

and the jboss-cxf-xxx.xml file needs to be set as

```
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:sec="http://cxf.apache.org/configuration/security"
        xmlns:http-conf="http://cxf.apache.org/transports/http/configuration"
        xsi:schemaLocation="http://cxf.apache.org/transports/http/configuration http://cxf.apache.org/schemas/c
onfiguration/http-conf.xsd http://www.springframework.org/schema/beans http://www.springframework.org/schema/be
ans/spring-beans-2.0.xsd http://cxf.apache.org/configuration/security http://cxf.apache.org/schemas/configurati
on/security.xsd">

       <http-conf:conduit name="*.http-conduit">
           <http-conf:authorization>
               <sec:AuthorizationType>Negotiate</sec:AuthorizationType>
               <sec:Authorization>MY_REALM</sec:Authorization>
           </http-conf:authorization>
       </http-conf:conduit>
</beans>
```

The resource adapter creation needs to define the following properties

```
   <config-property name="ConfigFile">path/to/jboss-cxf-xxxx.xml</config-property>
   <config-property name="ConfigName">test</config-property>
```

Note	Even though above configuration configures the value of "ConfigName", the cxf framework currently in the case of JAX-RS client does not give option to use it. For that reason use "*.http-conduit" which will apply to all the HTTP communications under this resource adapter.

Delegation

If in case the user is already logged into Teiid using Kerberos using JDBC/ODBC or used SPNEGO in web-tier and used pass-through authentication into Teiid, then there is no need to negotiate a new token for the Kerberos. The system can delegate the existing token.

To configure for delegation, set up security domain defined exactly as defined in "negotiation", and jboss-cxf-xxx.xml file, however remove the following line from jboss-cxf-xxx.xml file, as it is not going to negotiate new token.

```
<sec:Authorization>MY_REALM</sec:Authorization>
```

Add the following properties in web service resource adapter creation. One configures that "kerberos" security being used, the second defines a security domain to be used at the data source, in this case we want to use a security domain that passes through a logged in user

```
<config-property name="SecurityType">Kerberos</config-property>
<security>
     <security-domain>passthrough-security</security-domain>
</security>
```

To configure in "passthrough-security" security domain, the "security" subsystem add following XML fragment

```
<security-domain name="passthrough-security" cache-type="default">
    <authentication>
        <login-module code="Kerberos" flag="required" module="org.jboss.security.negotiation">
            <module-option name="delegationCredential" value="REQUIRED"/>
        </login-module>
    </authentication>
</security-domain>
```

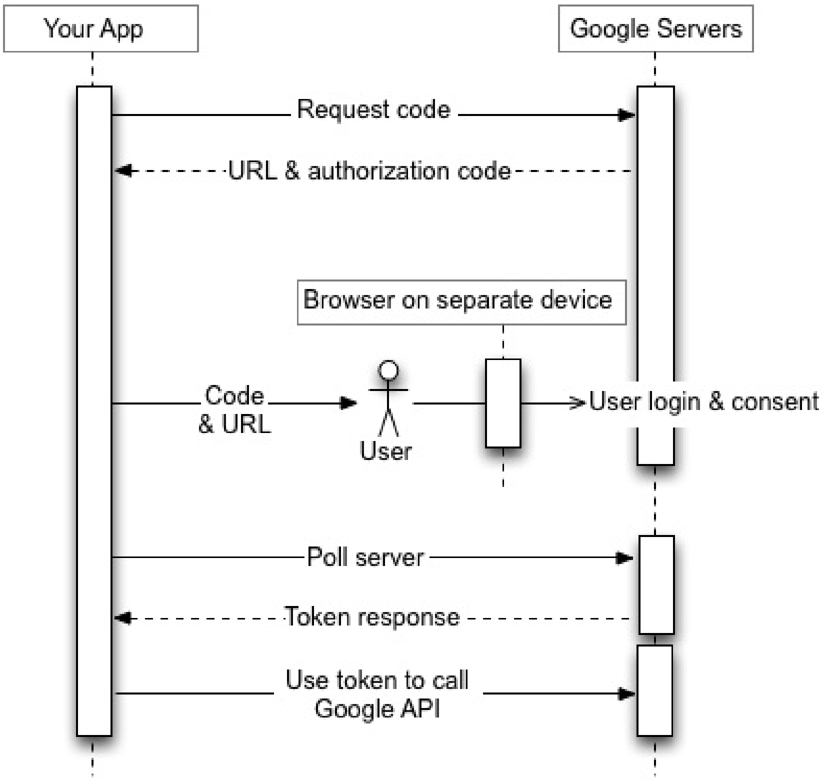
If in case there is no delegationCredential is available on the context, the access will fail.

OAuth Authentication With REST Based Services

Single user OAuth authentication

Web Services resource-adapter can be configured to participate in OAuth 1.0a and OAuth2 authentication schemes. Using Teiid along with "ws" translator and "web-services" resource adapter once write applications communicating with web sites like Google and Twitter.

In order to support OAuth authentication, there is some preparation and configuration work involved. Individual web sites typically provide developer facing REST based APIs for accessing their content on the web sites and also provide ways to register custom applications on user's behalf, where they can manage the Authorization of services offered by the web site. The first step is to register this custom application on the web site and collect consumer/API keys and secrets. The web-sites will also list the URLS, where to request for various different types of tokens for authorization using these credentials. A typical OAuth authentication flow is defined as below



The above image taken from https://developers.google.com/accounts/docs/OAuth2

To accommodate above defined flow, Teiid provides a utility called "teiid-oauth-util.sh" or "teiid-oauth-util.bat" for windows in the "bin" directory of your server installation. By executing this utility, it will ask for various keys/secrets and URLs for the generating the Access Token that is used in the OAuth authentication and in the end output a XML fragment like below.

```
$./teiid-oauth-util.sh
Select type of OAuth authentication
1) OAuth 1.0A
2) OAuth 2.0

2
=== OAuth 2.0 Workflow ===
```

```
Enter the Client ID = 10-xxxjb.apps.googleusercontent.com

Enter the Client Secret = 3L6-xxx-v9xxDlznWq-o

Enter the User Authorization URL = https://accounts.google.com/o/oauth2/auth

Enter scope (hit enter for none) = profile

Cut & Paste the URL in a web browser, and Authticate
Authorize URL  = https://accounts.google.com/o/oauth2/auth?client_id=10-
xxxjb.apps.googleusercontent.com&scope=profile&response_type=code&redirect_uri=urn%
3Aietf%3Awg%3Aoauth%3A2.0%3Aoob&state=Auth+URL

Enter Token Secret (Auth Code, Pin) from previous step = 4/z-RT632cr2hf_vYoXd06yIM-
xxxxx

Enter the Access Token URL = https://www.googleapis.com/oauth2/v3/token

Refresh Token=1/xxxx_5qzAF52j-EmN2U

Add the following XML into your standalone-teiid.xml file in security-domains
subsystem,
and configure data source securty to this domain

<security-domain name="oauth2-security">
    <authentication>
        <login-module code="org.teiid.jboss.oauth.OAuth20LoginModule"
flag="required" module="org.jboss.teiid.web.cxf">
            <module-option name="client-id" value="10-
xxxjb.apps.googleusercontent.com"/>
            <module-option name="client-secret" value="3L6-xxx-v9xxDlznWq-o"/>
            <module-option name="refresh-token" value="1/xxxx_5qzAF52j-EmN2U"/>
            <module-option name="access-token-uri"
value="https://www.googleapis.com/oauth2/v3/token"/>
        </login-module>
    </authentication>
</security-domain>
```

The XML fragment at the end defines the JAAS Login Module configuration, edit the standalone-teiid.xml and add it under "security-domains" subsystem. User needs to use this security-domain in their resource adapter as the security provider for this data source. An example resource-adapter configuration to define the data source to the web site in standalone-teiid.xml file looks like

```
<resource-adapter id="webservice3">
    <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:/googleDS" enabled="true" use-java-context="true" pool-name="teiid-ws-ds">
            <config-property name="SecurityType">
                OAuth
            </config-property>
```

```
            <security>
                <security-domain>oauth2-security</security-domain>
            </security>
        </connection-definition>
    </connection-definitions>
</resource-adapter>
---

Then, any query written using the "ws" translator and above resource-adapter will be automatically Authorized w
ith the target web site using OAuth, when you access a protected URL.

=== OAuth with Delegation

In the above configuration a single user is configured to access the web site, however if you want to delegate
logged in user’s credential as OAuth authentication, then user needs to extend the above LoginModule
_(org.teiid.jboss.oauth.OAuth20LoginModule or org.teiid.jboss.oauth.OAuth10LoginModule)_ and automate the proce
ss defined in the "teiid-oauth-util.sh" to define the Access Token
details dynamically. Since this process will be different for different web sites (it involves login and authen
tication), Teiid will not be able to provide single solution. However, user can extend the login module to prov
ide this feature much more easily since they will be working with targeted web sites.
```

VDB Versioning

VDB Versioning is a feature that allows multiple versions of a VDB to be deployed at the same time with additional support to determine which version will be used. If a specific version is requested, then only that VDB may be connected to. If no version is set, then the deployed VDBs are searched for the appropriate version. This feature helps support more fluid migration scenarios.

Version Property

When a user connects to Teiid the desired VDB version can be set as a connection property (See the Client Developer’s Guide) in JDBC or used as part of the VDB name for OData and ODBC access.

The vdb version is set in either the vdb.xxx or through a naming convention of the deployment name - vdbname.version.vdb, e.g. marketdata.2.vdb. The deployer is responsible for choosing an appropriate version number. If there is already a VDB name/version that matches the current deployment, then connections to the previous VDB will be terminated and its cache entries will be flushed. Any new connections will then be made to the new VDB.

Note	When setting the version in the vdb.xml or ddl file a unique deployment name must still be used as that is the name the application server internally uses for the deployment. Using the same deployment name as a previous version will simply overwrite the older deployment.

A simple integer version actually treated as the semantic version X.0.0. If desired a full semantic version can be used instead. A semantic version is up to three integers separated by periods.

Trailing version components that are missing are treated as zeros - version 1 is the same as 1.0.0 and version 1.1 is the same as 1.1.0.

JDBC and ODBC clients may use a version restriction - -vdbname.X. or vdbname.X.X. - note the trailing '.' which means a VDB that must match the partial version specified. For example vdbname.1.2. could match any 1.2.X version, but would not allow 1.3+ or 1.1 and earlier.

Connection Type

Once deployed a VDB has an updatable property called connection type, which is used to determine what connections can be made to the VDB. The connection type can be one of:

- **NONE**- disallow new connections.
- **BY_VERSION**- the default setting. Allow connections only if the version is specified or if this is the earliest BY_VERSION vdb and there are no vdbs marked as ANY.
- **ANY**- allow connections with or without a version specified.

The connection type may be changed either through the AdminConsole or the AdminAPI.

Deployment Scenarios

If only a select few applications are to migrate to the new VDB version, then a freshly deployed VDB would be left as BY_VERSION. This ensures that only applications that know the new version may use it.

If only a select few applications are to remain on the current VDB version, then their connection settings would need to be updated to reference the current VDB by its version. Then the newly deployed vdb would have its connection type set to ANY, which allows all new connections to be made against the newer version. If a rollback is needed in this scenario, then the newly deployed vdb would have its connection type set to NONE or BY_VERSION accordingly.

Logging

The Teiid system provides a wealth of information via logging. To control logging level, contexts, and log locations, you should be familiar with log4j and the container's *standalone-teiid.xml* or *domain-teiid.xml* configuration files depending upon the start up mode of WildFly.

All the logs produced by Teiid are prefixed by "org.teiid". This makes it extremely easy to control of of Teiid logging from a single context. Note however that changes to the log configuration file manually require a restart to take affect. CLI based log context modifications are possible, however details are beyond the scope of this document.

If you expect a high volume of logging information or use expensive custom audit/command loggers, it is a good idea to use an async appender to minimize the performance impact. For example you can use a configuration snippet like the one below to insert an async handler in front of the target appender.

```
<periodic-rotating-file-handler name="COMMAND_FILE">
    <level name="DEBUG" />
    <formatter>
        <pattern-formatter pattern="%d{HH:mm:ss,SSS} %-5p [%c] (%t) %s%E%n" />
    </formatter>
    <file relative-to="jboss.server.log.dir" path="command.log" />
    <suffix value=".yyyy-MM-dd" />
</periodic-rotating-file-handler>

<async-handler name="ASYNC">
  <level name="DEBUG"/>
  <queue-length value="1024"/>
  <overflow-action value="block"/>
  <subhandlers>
    <handler name="COMMAND_FILE"/>
  </subhandlers>
</async-handler>

<logger category="org.teiid.COMMAND_LOG">
    <level name="DEBUG" />
    <handlers>
        <handler name="ASYNC" />
    </handlers>
</logger>
```

Logging Contexts

While all of Teiid's logs are prefixed with *"org.teiid"*, there are more specific contexts depending on the functional area of the system. Note that logs originating from third-party code, including integrated org.jboss components, will be logged through their respective contexts and not through *"org.teiid"*. See the table below for information on contexts relevant to Teiid.

Context	Description
com.arjuna	Third-party transaction manager. This will include information about all transactions, not just those for Teiid.
org.teiid	Root context for all Teiid logs. Note: there are potentially other contexts used under org.teiid than are shown in this table.
org.teiid.PROCESSOR	Query processing logs. See also org.teiid.PLANNER for query planning logs.

org.teiid.PLANNER	Query planning logs.
org.teiid.SECURITY	Session/Authentication events - see also AUDIT logging
org.teiid.TRANSPORT	Events related to the socket transport.
org.teiid.RUNTIME	Events related to work management and system start/stop.
org.teiid.CONNECTOR	Connector logs.
org.teiid.BUFFER_MGR	Buffer and storage management logs.
org.teiid.TXN_LOG	Detail log of all transaction operations.
org.teiid.COMMAND_LOG	See command logging
org.teiid.AUDIT_LOG	See audit logging
org.teiid.ADMIN_API	Admin API logs.
org.teiid.ODBC	pg/ODBC logs.

Command Logging

Command logging captures executing commands in the Teiid System. This includes user commands (that have been submitted to Teiid at an INFO level), data source commands (that are being executed by the connectors at a DEBUG level), and query plans (at a TRACE level) are tracked through command logging.

The user command, "START USER COMMAND", is logged when Teiid starts working on the query for the first time. This does not include the time the query was waiting in the queue. And a corresponding user command, "END USER COMMAND", is logged when the request is complete (i.e. when statement is closed or all the batches are retrieved). There is only one pair of these for every user query.

The query plan command, "PLAN USER COMMAND", is logged when Teiid finishes the query planning process. There is no corresponding ending log entry, but with trace logging enabled the query plan will be included with subsequent user command events.

The data source command, "START DATA SRC COMMAND", is logged when a query is sent to the data source. And a corresponding data source command, "END SRC COMMAND", is logged when the execution is closed (i.e all the rows has been read). There can be one pair for each data source query that has been executed by Teiid, and there can be any number of pairs depending upon your user query.

The SRC command itself is then translated into 1 or more source statements, operations, etc. For sources that have textual representations of the native source query, each will be reported in a "SOURCE SRC COMMAND" event as at the DEBUG level with the field sourceCommand representing the SQL, SOQL, LDAP query etc. that is actually issued.

With this information being captured, the overall query execution time in Teiid can be calculated. Additionally, each source query execution time can be calculated. If the overall query execution time is showing a performance issue, then look at each data source execution time to see where the issue maybe.

To enable command logging to the default log location, simply enable the DETAIL level of logging for the org.teiid.COMMAND_LOG context.

To enable command logging to an alternative file location, configure a separate file appender for the DETAIL logging of the org.teiid.COMMAND_LOG context. An example of this is shown below and can also be found in the standalone-teiid.xml distributed with Teiid.

```
<periodic-rotating-file-handler name="COMMAND_FILE">
    <level name="DEBUG" />
    <formatter>
        <pattern-formatter pattern="%d{HH:mm:ss,SSS} %-5p [%c] (%t) %s%E%n" />
    </formatter>
    <file relative-to="jboss.server.log.dir" path="command.log" />
    <suffix value=".yyyy-MM-dd" />
</periodic-rotating-file-handler>

<logger category="org.teiid.COMMAND_LOG">
    <level name="DEBUG" />
    <handlers>
        <handler name="COMMAND_FILE" />
    </handlers>
</logger>
```

See the Developer's Guide to develop a custom logging solution if file based logging, or any other built-in Log4j logging, is not sufficient.

The following is an example of a data source command and what one would look like when printed to the command log:

```
2012-02-22 16:01:53,712 DEBUG [org.teiid.COMMAND_LOG] (Worker1_QueryProcessorQueue11 START DATA SRC COMMAND: st
artTime=2012-02-22 16:01:53.712
requestID=Ku4/dgtZPYk0.5 sourceCommandID=4 txID=null modelName=DTHCP translatorName=jdbc-simple sessionID=Ku4/d
gtZPYk0
principal=user@teiid-security
sql=HCP_ADDR_XREF.HUB_ADDR_ID, CPN_PROMO_HIST.PROMO_STAT_DT FROM CPN_PROMO_HIST, HCP_ADDRESS, HCP_ADDR_XREF
WHERE (HCP_ADDRESS.ADDR_ID = CPN_PROMO_HIST.SENT_ADDR_ID) AND (HCP_ADDRESS.ADDR_ID = HCP_ADDR_XREF.ADDR_ID) AND
(CPN_PROMO_HIST.PROMO_STAT_CD NOT LIKE 'EMAIL%') AND (CPN_PROMO_HIST.PROMO_STAT_CD <> 'SENT_EM') AND
(CPN_PROMO_HIST.PROMO_STAT_DT > {ts'2010-02-22 16:01:52.928'})
```

Note the following pieces of information:

- modelName: this represents the physical model for the data source that the query is being issued.
- translatorName: shows type of translator used to communicate to the data source.
- principal: shows the user account who submitted the query
- startTime/endTime: the time of the action, which is based on the type command being executed.
- sql: is the command submitted to the engine or to the translator for execution - which is NOT necessarily the final sql command submitted to the actual data source. But it does show what the query engine decided to push down.

END events will additionally contain:

- finalRowCount: the number of rows returned to the engine by the source query.
- cpuTime: the number of nanoseconds of cpu time used by the source command. Can be compared to the start/end wall clock times to determine cpu vs. idle time.

Audit Logging

Audit logging captures important security events. This includes the enforcement of permissions, authentication success/failures, etc.

To enable audit logging to the default log location, simply enable the DEBUG level of logging for the org.teiid.AUDIT_LOG context.

Additional Logging Information

Once a session has been created, each log made by Teiid will include the session id and vdb name/version in the MDC (mapped diagnostic context) with keys of teiid-session and teiid-vdb respectively.

Any log in the scope of a query will include the request id in the MDC with key of teiid-request.

Custom loggers, or format patterns, can take advantage of this information to better correlate log entries. See for example Teiid default *standalone-teiid.xml* that uses a pattern format which includes the session id prior to the message:

```
<pattern-formatter pattern="%d{HH:mm:ss,SSS} %-5p [%c] (%t) %X{teiid-session} %s%E%n"/>
```

Clustering in Teiid

Since Teiid is installed in WildFly, there is no additional configuration needed beyond what was performed when Teiid is setup in Domain Mode. See the Domain Mode section in the Teiid Installation Guide. Just make sure that you installed Teiid in every WildFly node and started all WildFly instances in the Domain mode that to be a part of the cluster.

Typically users create clusters to improve the performance of the system through:

- **Load Balancing**: Take look at HAProxy below and in the Client Developer’s Guide on how to use simple load balancing between multiple nodes.
- **Fail Over**: Take look at the Client Developer’s Guide on how to use fail over between multiple nodes.
- **Distributed Caching**: This is automatically done for you once you configure it as specified above.
- **Event distribution**: metadata and data modifications will be distributed to all cluster members.

In the *Domain* mode, the only way a user can deploy any artifacts is using either CLI or using the Admin API. Copying VDB directly into the "deployments" directory is not supported.

Note	some load balancers have timeouts that cannot be adjusted. You may need to adjust the tcp_keepalive_time on your client OS. The default is typically 2 hours, which is much too long in many cases. See Custom Configuration of TCP Socket Keep-Alive Timeouts.

HAProxy

HAProxy may be used for load-balancing and high availability. A good tutorial is located at Load Balancing JDV-HAProxy or see Luigi Fugaro’s example.

The load balancer should use an algorithm that supports sticky connections as Teiid sessions as specific to the original host. For HAProxy it is recommended that you use leastconn or source.

Monitoring

Teiid provides information about its current operational state. This information can be useful in tuning, monitoring, and managing load and through-put. The runtime data can be accessed using administrative tools (i.e. Admin Console or Admin API).

Query/Session details:

Name	Description
Current Sessions	List current connected sessions
Current Request	List current executing requests
Current Transactions	List current executing transactions
Query Plan	Retrieves the query plan for a specific request

There are administrative options for terminating sessions, queries, and transactions.

Metrics:

Session/Query

Name	Property	Description	Comment
Session Count	sessionCount	Indicates the number of user connections currently active	To ensure number of sessions are not restricted at peak times, check *max-sessions-allowed* (default 10000) is set accordingly and review *sessions-expiration-timelimit*
Query Count	queryCount	Indicates the number of queries currently active.	
Active Query Plan Count	ENGINE_STATISTIC.active-plans-count	Number of query plans currently being processed	To ensure maximum through-put, see the QueryEngine section in Threading on tuning.
Waiting Query Plan Count	ENGINE_STATISTIC.waiting-plans-count	Number of query plans currently waiting	
Max Waiting Query Plan Watermark	ENGINE_STATISTIC.max-waitplan-watermark	The maximum number of query plans that have been waiting at one time, since the last time the server started	

Long Running Queries	longRunningQueries	List current executing queries that have surpassed the query threshold(*query-threshold-in-seconds*).	Setup alert to warn when one or more queries are consuming resources for an extended period of time. If running too long, an option is to cancel request or increase threshold.

Buffer Manager

For tuning suggestions, see Memory Management.

Name	Property	Description	Comment
Disk Write Count	ENGINE_STATISTIC.buffermgr-disk-write-count	Disk write count for the buffer manager.	
Disk Read Count	ENGINE_STATISTIC.buffermgr-disk-read-count	Disk read count for the buffer manager.	
Cache Write Count	ENGINE_STATISTIC.buffermgr-cache-write-count	Cache write count for the buffer manager.	
Cache Read Count	ENGINE_STATISTIC.buffermgr-cache-read-count	Cache read count for the buffer manager.	
Disk Space Used (MB)	ENGINE_STATISTIC.buffermgr-diskspace-used-mb	Indicates amount of storage space currently used by buffer files	Setup alert to warn when used buffer space is at an unacceptable level, based on the setting of *max-buffer-space*
Total memory in use (KB)	ENGINE_STATISTIC.total-memory-inuse-kb	Estimate of the current memory usage in kilobytes.	
Total memory in use by active plans (KB)	ENGINE_STATISTIC.total-memory-inuse-active-plans-kb	Estimate of the current memory usage by active plans in kilobytes	

Plan/Result Cache

For tuning suggestions, see Cache Tuning.

Name	Property	Description
Prepared Plan Cache Size	PREPARED_PLAN_CACHE.total-entries	Current number of entries in cache.
Prepared Plan Cache # of Requests	PREPARED_PLAN_CACHE.request-count	Total number of requests made against cache.
Prepared Plan Cache Hit Ratio %	PREPARED_PLAN_CACHE.hit-ratio	Percentage of positive cache hits
ResultSet Cache Size	QUERY_SERVICE_RESULT_SET_CACHE.total-entries	Current number of entries in cache.

ResultSet Cache # of Requests	QUERY_SERVICE_RESULT_SET_CACHE.request-count	Total number of requests made against cache.
ResultSet Cache Hit Ratio %	QUERY_SERVICE_RESULT_SET_CACHE.hit-ratio	Percentage of positive cache hits.

Performance Tuning

Performance tuning can be done by changing the property settings defined in the teiid subsystem and its sub components.

Execute the following command on CLI to see the possible settings at the root of the teiid subsystem:

```
/subsystem=teiid:read-resource-description
```

There are several categories of properties:

1. Memory Management
2. BufferManager: all properties that start with "buffer-service"
3. Cache Tuning: all properties that start with "resultset-cache" or "preparedplan-cache"
4. Threading
5. LOBs
6. Other Considerations

Socket Transport settings for one of the supported transport types (i.e., jdbc, odbc, embedded) can be viewed by executing the following command:

```
/subsystem=teiid/transport={jdbc | odbc | embedded}:read-resource-description
```

Memory Management

The *BufferManager* is responsible for tracking both memory and disk usage by Teiid. Configuring the *BufferManager* properly along with data sources and threading ensures high performance. In most instances though the default settings are sufficient as they will scale with the JVM and consider other properties such as the setting for max active plans.

Execute following command on CLI to find all possible settings on *BufferManager*:

```
/subsystem=teiid:read-resource
```

All the properties that start with "buffer-manager" used to configure *BufferManager*. Shown below are the CLI write attribute commands to change *BufferManager's* settings (all show the default setting):

```
/subsystem=teiid:write-attribute(name=buffer-manager-inline-lobs,value=true)
/subsystem=teiid:write-attribute(name=buffer-manager-processor-batch-size,value=256)

/subsystem=teiid:write-attribute(name=buffer-manager-heap-max-processing-kb,value=-1)
/subsystem=teiid:write-attribute(name=buffer-manager-heap-max-reserve-mb,value=-1)

/subsystem=teiid:write-attribute(name=buffer-manager-storage-enabled,value=true)
/subsystem=teiid:write-attribute(name=buffer-manager-storage-max-object-size-kb,value=8196)

/subsystem=teiid:write-attribute(name=buffer-manager-fixed-memory-buffer-space-mb,value=-1)
/subsystem=teiid:write-attribute(name=buffer-manager-fixed-memory-buffer-off-heap,value=false)

/subsystem=teiid:write-attribute(name=buffer-manager-disk-max-space-mb,value=51200)
/subsystem=teiid:write-attribute(name=buffer-manager-disk-encrypt-files,value=false)
/subsystem=teiid:write-attribute(name=buffer-manager-disk-max-open-files,value=64)
/subsystem=teiid:write-attribute(name=buffer-manager-disk-max-file-size-mb,value=2048)
```

Note	It is not recommend that to change these properties until there is an understanding of the properties (elaborated below) and any potential issue that is being experienced.

Some of *BufferManager's* properties are described below. Note that the performance tuning advice is highlighted in info boxes.

General Properties

processor-batch-size (default 256) - Specifies the target row count of a batch of the query processor. A batch is used to represent both linear data stores, such as saved results, and temporary table pages. Teiid will adjust the processor-batch-size to a working size based upon an estimate of the data width of a row relative to a nominal expectation of 2KB. The base value can be doubled or halved up to three times depending upon the data width estimation. For example a single small fixed width (such as an integer) column batch will have a working size of **processor-batch-size * 8** rows. A batch with hundreds of variable width data (such as string) will have a working size of **processor-batch-size / 8** rows. Any increase in the processor batch size beyond the first doubling should be accompanied with a proportional increase in the max-storage-object-size to accommodate the larger storage size of the batches.

Note	Additional considerations are needed if large VM sizes and/or datasets are being used. Teiid has a non-negligible amount of overhead per batch/table page on the order of 100-200 bytes. If you are dealing with datasets with billions of rows and you run into memory issues, then after examining the root cause if you see that it's solely related to memory held by a significant number of batch references, then consider increasing the *processor-batch-size* to force the allocation of larger batches and table pages. A general guideline would be to double processor-batch-size for every doubling of the effective heap for Teiid beyond 4 GB - processor-batch-size = 512 for an 8 GB heap, processor-batch-size = 1024 for a 16 GB heap, etc.

inline-lobs (default true) - Small lobs will be stored in their batch directly rather than managed out of band. Should generally be left as true to minimize the fetch costs of small lobs.

Heap Properties

The amount of estimated heap in direct object references to batches / pages held by the *BufferManager* can be adjusted.

heap-max-reserve-mb (default -1) - setting determines the total size in kilobytes of batches that can be held by the *BufferManager* in memory. This number does not account for persistent batches held by soft (such as index pages) or weak references. The default value of -1 will auto-calculate a typical max based upon the max heap available to the VM. The auto-calculated value assumes a 64bit architecture and will limit buffer usage to 40% of the first gigabyte of memory beyond the first 300 megabytes (which are assumed for use by the AS and other Teiid purposes) and 50% of the memory beyond that. The additional caveat here is that if the size of the memory buffer space is not specified, then it will effectively be allocated out of the max reserve space. A small adjustment is also made to the max reserve to account for batch tracking overhead.

Note	With default settings and an 8GB VM size[*], then *heap-max-reserve-mb* will be: (1024-300) * 0.4) + (7 * 1024 * 0.5 = 4373.6 MB before considering the overhead for persistent batches or the fixed memory buffer. The fixed memory buffer will by default use 40% of that initial calculation. Once additional overhead is removed, the actual heap-max-reserve-mb will be around 2624 MB.

[*] Teiid will use the max memory reported by the runtime. This value may be lower than the Xmx setting used as a VM argument as the VM will adjust for necessary overheads.

The *BufferManager* automatically triggers the use of a canonical value cache if enabled when more than 25% of the reserve is in use. This can dramatically cut the memory usage in situations where similar value sets are being read through Teiid, but does introduce a lookup cost. If you are processing small or highly similar datasets through Teiid, and wish to conserve memory, you should consider enabling value caching.

Warning	Memory consumption can be significantly more or less than the nominal target depending upon actual column values and whether value caching is enabled. Large non built-in type objects can exceed their default size estimate. If an out of memory errors occur, then set a lower *heap-max-reserve-mb* value. Also note that source lob values are held by memory references that are not cleared when a batch is persisted. With heavy lob usage you should ensure that buffers of other memory associated with lob references are appropriately sized.

heap-max-processing-kb (default -1) - setting determines the total size in kilobytes of batches that can be guaranteed for use by *one* active plan and may be in addition to the memory held based on *heap-max-reserve-mb*. Typical minimum memory required by Teiid when all the active plans are active is *#active-plans*heap-max-processing-kb*. The default value of -1 will auto-calculate a typical max based upon the max heap available to the VM and max active plans. The auto-calculated value assumes a 64bit architecture and will limit nominal processing batch usage to less than 10% of total memory.

Note	With default settings including 20 active-plans and an 8GB VM size, then *heap-max-processing-kb* will be: (.07 * 8 * 1024)/20^.8 = 537.4 MB/11 = 52.2 MB or 53,453 KB per plan. This implies a nominal range between 0 and 1060 MB that may be reserved with roughly 53 MB per plan. You should be cautious in adjusting heap-max-processing-kb. Typically it will not need adjusted unless you are seeing situations where plans seem memory constrained with low performing large sorts.

Storage Properties

The tiers of memory below the heap hold the batches / pages in a denser serialized columnar form. The lowest level is disk storage. Fronting disk is a fixed memory buffer, which can be allocated on or off heap, that acts as a serialization buffer and cache for reads/writes to disk.

storage-enabled (default true) - If disabled, batches / pages that are pushed to the storage layer are instead held in memory. Also all temporary lob space will be allocated from memory as well. Generally only useful in constrained or testing situations.

storage-max-object-size-kb (default 8196 or 8MB) - The maximum size of a buffered managed object in bytes and represents the individual batch page size. If the *processor-batch-size* is increased and/or you are dealing with extremely wide result sets (several hundred columns), then the default setting of 8MB for the *max-storage-object-size* may be too low. The inline-lobs setting also can increase the size of batches containing small lobs. The sizing for *max-storage-object-size* is in terms of serialized size, which will be much closer to the raw data size than the Java memory footprint estimation used for *max-reserved-mb*. *max-storage-object-size* should not be set too large relative to *memory-buffer-space* since it will reduce the performance of the memory buffer. The memory buffer supports only 1 concurrent writer for each *max-storage-object-size* of the *memory-buffer-space*. Note that this value does not typically need to be adjusted unless the processor-batch-size is adjusted, in which case consider adjusting it in proportion to the increase of the processor-batch-size.

Note	If exceptions occur related to missing batches and "*TEIID30001 Max block number exceeded*" is seen in the server log, then increase the storage-max-object-size-kb to support larger storage objects. Alternatively you could make the processor-batch-size smaller.

Fixed Memory Properties

fixed-memory-buffer-space-mb (default -1) - This controls the amount of on or off heap memory allocated as byte buffers for use by the Teiid buffer manager measured in megabytes. This setting defaults to -1, which automatically determines a setting based upon whether it is on or off heap and the value for *heap-max-reserve-mb*. The memory buffer supports only 1 concurrent writer for each *storage-max-object-size-mb* of the *fixed-memory-buffer-space-mb*. Any additional space serves as a cache for the serialized for of batches.

Note	When left at the default setting the calculated memory buffer space will be approximately 40% of the *heap-max-reserve-mb* size. If the memory buffer is on heap and the *heap-max-reserve-mb* is automatically calculated, then the memory buffer space will be subtracted out of the effective *heap-max-reserve-mb*. If the memory buffer is off heap and the *heap-max-reserve-mb* is automatically calculated, then it’s size will be reduced slightly to allow for effectively more working memory in the vm.

fixed-memory-buffer-off-heap (default false) - Setting *fixed-memory-buffer-off-heap* to "true" will allocate the Teiid memory buffer off heap. Depending on whether your installation is dedicated to Teiid and the amount of system memory available, this may be preferable to on-heap allocation. The primary benefit is additional memory usage for Teiid without additional garbage collection tuning. This becomes especially important in situations where more than 32GB of memory is desired for the VM. Note that when using off-heap allocation, the memory must still be available to the java process and that setting the value of *memory-buffer-space* too high may cause the VM to swap rather than reside in memory. With large off-heap buffer sizes (greater than several gigabytes) you may also need to adjust VM settings.

Note	**Oracle/Sun VM** - the relevant VM settings are MaxDirectMemorySize and UseLargePages. For example adding: '-XX:MaxDirectMemorySize=12g -XX:+UseLargePages' to the VM process arguments would allow for an effective allocation of approximately an 11GB Teiid memory buffer (the **fixed-memory-buffer-space-mb** setting) accounting for any additional direct memory that may be needed by the AS or applications running in the AS.

Disk Properties

disk-max-space-mb (default 51200) - For table page and result batches the buffer manager will have a limited number of files that are dedicated to a particular storage size. However, as mentioned in the installation, creation of Teiid lob values (for example through SQL/XML) will typically create one file per lob once the lob exceeds the allowable in memory size of 32KB. In heavy temporary lob usage scenarios, consider pointing the buffer directory on a partition that is routinely defragmented. By default Teiid will use up to 50GB of disk space. This is tracked in terms of the number of bytes written by Teiid. For large data sets, you may need to increase the *disk-max-space-mb* setting.

disk-max-file-size-mb (default 2048) - Each intermediate result buffer, temporary LOB, and temporary table is stored in its own set of buffer files, where an individual file is limited to *disk-max-file-size-mb* megabytes. Consider increasing the storage space available to all such files by increasing *disk-max-space-mb,* if your installation makes use of internal materialization, makes heavy use of SQL/XML, or processes large row counts.

Limitations

It's also important to keep in mind that Teiid has memory and other hard limits which breaks down along several lines in terms of # of storage objects tracked, disk storage, streaming data size/row limits, etc.

1. The buffer manager has a max addressable space of 16 terabytes - but due to fragmentation you'd expect that the max usable would be less. This is the maximum amount of storage available to Teiid for all temporary lobs, internal tables, intermediate results, etc.

2. The max size of an object (batch or table page) that can be serialized by the buffer manager is 32 GB - but you should approach that limit (the default limit is 8 MB). A batch/page is set or rows that are flowing through Teiid engine and is dynamically scaled based upon the estimated data width so that the expected memory size is consistent.

3. The heap-max-processing-kb and heap-max-reserve-mb are based upon memory footprint estimations and not exact sizes - actual memory usage and garbage collection cycles are influenced by a lot of other factors.

4. The maximum row count for any interface, JDBC/ODBC/OData, is 2^31-1 rows.

Handling a source that has tera/petabytes of data doesn't by itself impact Teiid in any way. What matters is the processing operations that are being performed and/or how much of that data do we need to store on a temporary basis in Teiid. With a simple forward-only query, Teiid will return a petabytes of data with minimal memory usage.

Other Limits

To prevent run away memory or disk consumption:

1. Error code TEIID31260. A single lob (xml, clob, blob, json) created on the server side is limited to the .25 * (max buffer space) / (max active plans).

2. Error code TEIID31261. A single table or tuple buffer is limited to a portion of the total max reserve, fixed memory buffer, and disk space.

If needed an administrator can further restrict memory usage from each session by setting the system property org.teiid.maxSessionBufferSizeEstimate to the desired size in bytes. This is based upon the memory footprint estimate and may not correspond exactly to heap or disk consumption.

Other Considerations for Sizing

Each batch/table page requires an in memory cache entry of approximately ~ 128 bytes - thus the total tracked max batches are limited by the heap and is also why we recommend to increase the processing batch size on larger memory or scenarios making use of large internal materializations. The actual batch/table itself is managed by buffer manager, which has layered memory buffer structure with spill over facility to disk.

Using internal materialization is based on the BufferManager. BufferManager settings may need to be updated based upon the desired amount of internal materialization performed by deployed vdbs.

If an out of memory error occurs it is best to first capture a heap dump to determine where memory is being held - tweaking the BufferManager settings may not be necessary depending upon the cause.

Common Configuration Scenarios

In addition to scenarios outlined above, a common scenario would be to minimize the amount of on heap space consumed by Teiid. This can be done by moving the memory buffer to off heap with the fixed-memory-buffer-off-heap setting or by restricting the heap-max-reserve-mb setting. Reducing the heap-max-processing-kb setting should generally not be necessary, unless there is a need to severely restrict the heap usage beyond the heap-max-reserve-mb setting.

Transport

max-socket-threads (default 0) - The max number of threads dedicated to the initial request processing. Zero indicates to use the system default of max available processors. All the access to Teiid (JDBC, ODBC, etc) is controlled by "transport" element in the configuration. Socket threads are configured for each transport. They handle NIO non-blocking IO operations as well as directly servicing any operation that can run without blocking. For longer running operations, the socket threads queue with work the query engine.

Query Engine

max-threads (default 64) - The query engine has several settings that determine its thread utilization. `max-threads` sets the total number of threads available in the process pool for query engine work (processing plans, transaction control operations, processing source queries, etc.). You should consider increasing the maximum threads on systems with a large number of available processors and/or when it's common to issue non-transactional queries that issue a large number of concurrent source requests.

max-active-plans (default 20) - Should always be smaller than *max-threads*. By default, thread-count-for-source-concurrency is calculated by (*max-threads* / *max_active_plans*) * 2 to determine the threads available for processing concurrent source requests for each user query. Increasing the *max-active-plans* should be considered for workloads with a high number of long running queries and/or systems with a large number of available processors. If memory issues arise from increasing the *max-threads* and *max-active-plans*, then consider decreasing the amount of heap held by the buffer manager or decreasing the *processor-batch-size* to limit the base number of memory rows consumed by each plan.

thread-count-for-source-concurrency (default 0) - Should always be smaller than *max-threads*, sets the number of concurrently executing source queries per user request. 0 indicates to use the default calculated value based on 2 * (*max-threads* / *max-active-plans*). Setting this to 1 forces serial execution of all source queries by the processing thread. Any number greater than 1 limits the maximum number of concurrently execution source requests according. Using the respective defaults, this means that each user request would be allowed 6 concurrently executing source queries. If the default calculated value is not applicable to your workload, for example, if you have queries that generate more concurrent long running source queries, you should adjust this value.

time-slice-in-milliseconds (default 2000) - Provides course scheduling of long running processor plans. Plans whose execution exceed a time slice will be re-queued for additional processing to allow for other plans to be initiated. The time slice is from the perspective of the engine processing thread. This value is not honored exactly as the plan may not be at a re-startable point when the time slice expires. This is not a replacement for the thread scheduling performed by Java and the operating system, rather it just ensures that Teiid allows other work to be started if the current set of active plans includes long running queries.

Async Operations

async-thread-pool-max-thread-count (default 10) - Controls the number of threads available for system level async operations, such as metadata load.

Cache Tuning

Caching can be tuned for cached results (including user query results and procedure results) and prepared plans (including user and stored procedure plans). Even though it is possible to disable or otherwise severely constrain these caches, this would probably never be done in practice as it would lead to poor performance.

Cache statistics can be obtained through the Admin Console or the AdaminAPI. The statistics can be used to help tune cache parameters and ensure a hit ratio.

Plans are currently fully held in memory and may have a significant memory footprint. When making extensive use of prepared statements and/or virtual procedures, the size of the plan cache may be increased proportionally to number of gigabytes intended for use by Teiid.

While the result cache parameters control the cache result entries (max number, eviction, etc.), the result batches themselves are accessed through the BufferManager. If the size of the result cache is increased, you may need to tune the BufferManager configuration to ensure there is enough buffer space.

Result set and prepared plan caches have their entries invalidated by data and metadata events. By default these events are captured by running commands through Teiid. See the Developers Guide for further customization. Teiid stores compiled forms of update plans or trigger actions with the prepared plan, so that if metadata changes, for example by disabling a trigger, changes may take effect immediately. The default *max-staleness* for result set caching is 0 seconds or immediate invalidation. Consider increasing this value to increase result set cache hits. Even with a setting of 0, full transactional consistency is not guaranteed - rather the underlying Infinispan cache must be configured with a transaction mode of XA.

Socket Transports

Teiid separates the configuration of its socket transports for JDBC and pg/ODBC. You have the option of also configuring secure versions of these transports. Typical installations will not need to adjust the default thread and other low level settings.

The default values for *input-buffer-size* and *output-buffer-size* are set to 0, which will use the system default. Before adjusting these values, keep in mind that each JDBC/ODBC connection will create a new socket. Setting these values to a large buffer size should only be done if the number of clients are constrained. All JDBC/ODBC socket operations are non-blocking, so setting the number of *max-socket-threads* higher than the maximum effective parallelism of the machine should not result in greater performance. The default value 0 indicates the system default of 2 * available processors will be used.

Note	If you are using more than the 2 default socket transports on a machine with a high number of actual or virtual cores, you may need to consider manually configuring the max threads for each to transport to cut down on the number of threads created.

JDBC clients may need to adjust low-level transport values, in addition to SSL Client Connection properties via a teiid-client-settings.properties file. This file also contains buffer, socket pooling, and maxObjectSize (effectively the maximum response size) settings.

LOBs

LOBs and XML documents are streamed from the Teiid Server to the Teiid JDBC API.   Normally, these values are not materialized in the server memory - avoiding potential out-of-memory issues. When using style sheets and non-streaming XQuery whole XML documents must be materialized on the server. Even when using the XMLQuery or XMLTable functions and document projection is applied, memory issues may occur for large documents.

LOBs are broken into pieces when being created and streamed. The maximum size of each piece when fetched by the client can be configured with the *"lob-chunk-size-in-kb"* property on Teiid configuration. The default value is 100 KB. When dealing with extremely large LOBs, you may consider increasing this value to decrease the amount of round-trips to stream the result. Setting the value too high may cause the server or client to have memory issues.

Source LOB values (LOBs from physical sources) are typically accessed by reference, rather than having the value copied to a temporary location. Thus care must be taken to ensure that source LOBs are returned in a memory-safe manner. This caution is more for the source driver vendors to not to consume VM memory for LOBs. Translators via the copyLobs property can instead copy lob values to a temporary location.

Cached lobs will be copied rather than relying on the reference to the source lob.

Temporary lobs created by Teiid will be cleaned up when the result set or statement is closed. To rely on implicit garbage collection based cleanup instead of statement close, the Teiid session variable clean_lobs_onclose can be set to false (by issuing the query "SELECT teiid_session_set('clean_lobs_onclose', false)" - which can be done for example via the new connection sql in the datasource definition). This can be used for local client scenarios that relied on the implicit behavior.

Other Considerations

When using Teiid in a development environment, you may consider setting the *max-source-rows-allowed* property to reasonably small level value (e.g. 10000) to prevent large amounts of data from being pulled from sources. Leaving the *exception-on-max-source-rows* set to "true" will alert the developer through an exception that an attempt was made to retrieve more than the specified number of rows.

Teiid Console

Teiid Console is a web based administrative and monitoring tool for Teiid. Teiid Console is extension of WildFly console that is built using GWT based technologies. There are two primary Teiid kits - an overlay for an existing WildFly install, and an all in one that includes the WildFly server and Teiid console.

The Web Console is now maintenance only. New work related to web tooling will be alligned with OpenShift efforts.

Installation

If you start with just the overlay, you may separately install the Teiid Console. Unzip the contents over the WildFly root directory and all the required files will be overlayed correctly to install Teiid Console. See all downloads on teiid.io.

Management User

The Teiid Console, by default is secured, so you would need a management realm user id and password to log in. In the <install>/bin directory, use

Adding a management user in linux

```
./add-user.sh
```

Adding a management user in Windows

```
add-user.bat
```

then follow the prompts to create Management Realm user. Once you have created a management user, you need to use these credentials to log in to the Teiid Console.

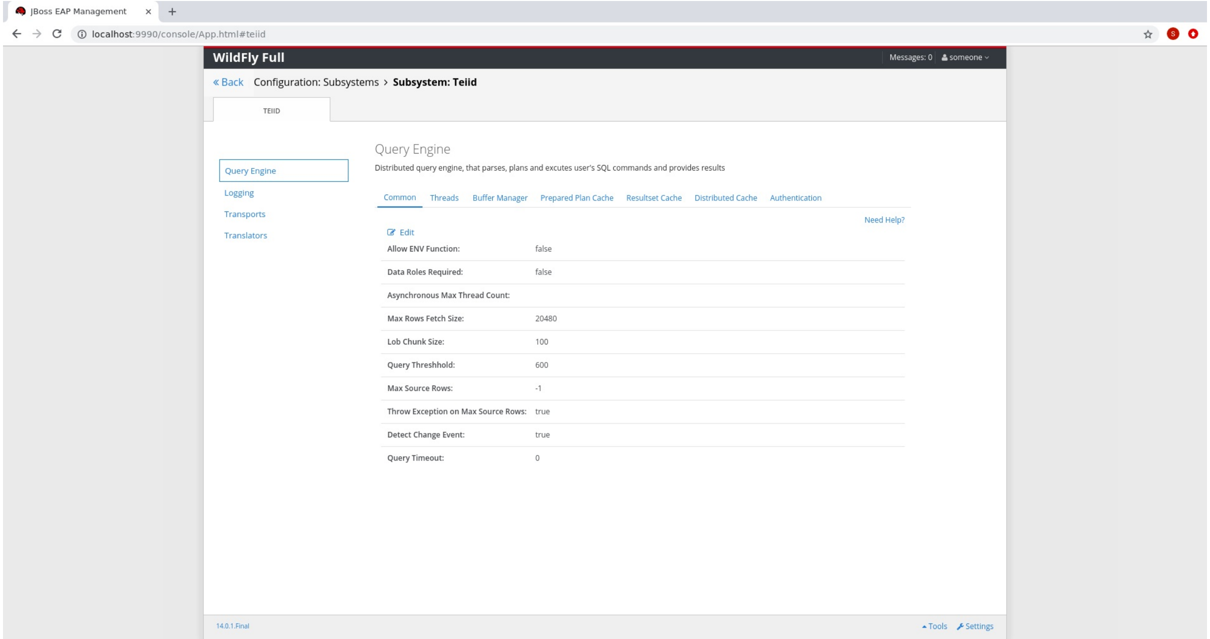
Accessing The Console

If you have started your WildFly in default mode, then you can access the Teiid Console at http://localhost:9990/console/App.html. If you have altered the binding or port numbers then modify the address accordingly.

Configuration

Click on the configuration tab at the top of the main console screen. Under Subsystems click on "Teiid" in left navigation tree. There you have four choices:

- Query Engine - view and configure core Teiid engine properties.
- Translators - view, add and remove the Translators configured in Teiid.
- Transports - view, add and remove transports to the Teiid engine.
- Logging - toggle command / audit / trace logging.

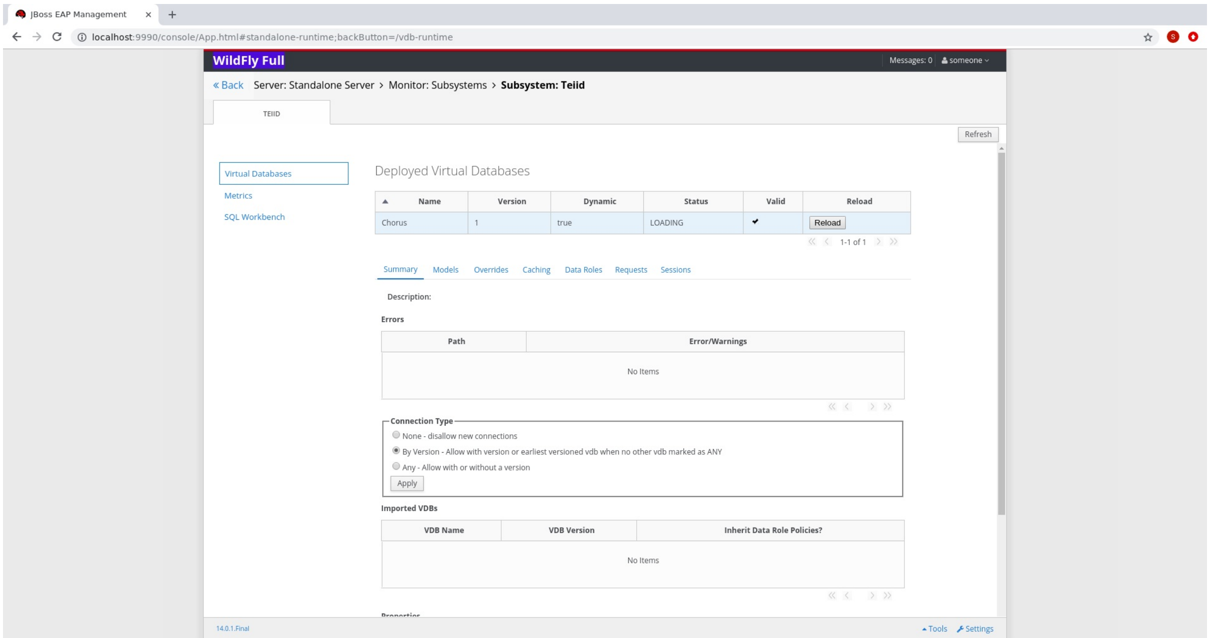


Using this view you can change any configuration value of Teiid. Note that various different configuration properties are sub-divided into different tabs. You can click "Need Help" link on these pages to see the description of each field on the page.

Note	**Server Restart** - some properties require you to restart the server before they can take effect.

Runtime View

Runtime view shows runtime information about WildFly and the subsystems including Teiid. Click on the Runtime tab, select the Standalone Server (or whatever server is appropriate), select Subsystems, then Teiid.



Using this page user can view many different settings in the context a VDB. All the VDBs deployed in the server are shown in top level table. When you select and highlight a VDB, more details about that VDB are displayed in the sub-tabs below. Each of these sub-tabs are divided into grouping of the functionality.

Summary

This tab shows the description and any properties associated with VDB, along with any other VDBs this VDB imports. This tab is designed to give a quick overview of the VDB status.

Models

This panel shows all the models that are defined in a given VDB, and shows each models translator name and source connection JNDI name. It also shows the type of models and if it is multi-source or not. When a particular model is selected it will show all properties of that model that are defined and also shows any errors associated with the model. When your VDB is not deployed in the "active" status, you would need to verify these errors and fix to resolve any deployment issues.

The "DDL" button shows the schema for the given model.

The tool lets the user edit the translator name or JNDI name by double clicking on them and modifying them. This useful if you would like to change the JNDI name in a given environment.

Overrides

If you have overridden any translators this panel will show the all the overridden translators and their properties.

Caching

Caching panel shows caching statistics of resultset cache as to how effectively the cache is being used. It also shows all the internal materialized views in the VDB and their load status as to when they were loaded. It also gives options to invalidate a specific view or all the views in a VDB, so that they can refresh/reload the contents from source.

This panel also provides a UI to flush the entire the resultset cache contents or prepared plan cache contents for the selected VDB.

Data Roles

Data Roles panel shows the all the policies that defined in the VDB. For each selected policy, it will also list the "permissions" for that policy as to what kind of authorizations user has and shows the mapped enterprise role assignments to that policy. You can even add/remove a enterprise role to the policy using the this UI.

Requests

This panel shows all the current requests against the selected VDB at the time of VDB selection. You can click "refresh" to get a more up to date requests. The top table in the panel shows the user submitted requests to the teiid engine, when one of those requests are selected, then the bottom table shows all the source level queries that are submitted to the physical sources by Teiid engine.

Using this UI, user can also submit a "cancel" request to a user level query. Since "cancel" asynchronous operation, the operation is not guaranteed as query may already been finished, by the time cancel is submitted.

Sessions

This panel shows all the active sessions that are connected to the selected VDB. It shows their connection properties and also gives an option to terminate either a selected session or all the sessions.

FAQ

- *How to deploy a VDB in standalone mode?*

In the `Deployments` view, click `add` and select the VDB to deploy. Also make sure you `enable` the VDB once it is deployed.

- *How to create Data source?*

In the `Configuration` view, go to `Subsystem` → `Datasources` → `XA/Non-XA`, click `add` and follow the wizard to create JDBC data source.

If you trying to create connection to Teiid based File, Salesforce or WS based connections, select `Subsystem` → `Resource Adaptors` and click `add`.

- *How to add COMMAND Logging?*

In the `Configuration` view, go to `Subsystem` → `Logging`, click view, on `Log Categories` tab, click add `org.teiid.COMMAND_LOG` in `DEBUG` mode. The default log will be in the FILE handler. You can even add other handler if choose to do so.

- *Change Teiid JDBC Port in standalone mode?*

In the `Configuration` view, go to `Socket Binding` click `View`, view the `standard-sockets` select `teiid-jdbc` and edit.

System Properties and Environment Variables

Some of Teiid's low-level behavior can be configured via system or env properties, rather than through configuration files.

A typical place to set system properties for WildFly launches is in the <install>/bin/<mode>.conf. A property setting has the format `-Dproperty=value`.

With 13.0 environment variables will be checked after the corresponding system property. This allows for Teiid client and server code running in Docker or on OpenShift to be easily configured. The environment property key will be checked by converting it first to upper snake case - which replaces lower case with upper case, any period with _ and separates words with _. For example org.teiid.allowNanInfinity would check the environment key ORG_TEIID_ALLOW_NAN_INFINITY.

Table of Contents

General

Setting	Description	Default Value
org.teiid.allowNanInfinity	Set to true to allow numeric functions to return NaN (Not A Number) and +-Infinity. Note that these values are not covered by the SQL specification.	false
org.teiid.useValueCache	Set to true to enable the canonical value cache. Value caching is used dynamically when buffer memory is consumed to reuse identical values and thus reduce the memory consumed by Teiid. There is a computation cost associated with the cache lookup, so enabling this setting is not appropriate for installations handling large volumes of dissimilar data.	false
org.teiid.ansiQuotedIdentifiers	Set to false to emulate Teiid 6.x and prior behavior of treating double quoted values without leading identifier parts as string literals, which is not expected by the SQL specification.	true
org.teiid.subqueryUnnestDefault	If true, the optimizer will aggressively unnest subqueries in WHERE predicates. If possible the predicate will be unnested to a traditional join and will be eligible for dependent join planning.	false
org.teiid.ODBCPacketSize	Target size in bytes of the ODBC results buffer. This is not a hard maximum, lobs and wide rows may use larger buffers.	307200

org.teiid.decimalAsDouble	Set to true to parse exact fixed point literals, e.g. 1.0, as double values rather than as decimal/BigDecimal values and to return a double value from the AVG function for integral values in the same way as releases earlier than 8.0.	false
org.teiid.comparableLobs	Set to true to allow blob and clob column values to be comparable in Teiid. Source type metadata will determine if the comparison can be pushed down.	false
org.teiid.comparableObject	Set to true to allow object column values to be comparable in Teiid. Source type metadata will determine if the comparison can be pushed down. The object instances are expected to correctly implement java.lang.Comparable.compareTo. If the instance object is not Comparable, then ClassCastExceptions may the thrown.	false
org.teiid.padSpace	Set to true to compare strings as if PAD SPACE collation is being used, that is strings are effectively right padded to the same length for comparison. If this property is set, it is not necessary to use the trimStrings translator option.	false
org.teiid.collationLocale	Set to a Java locale string language[_country[_varient]], where language, country, and variant are two letter codes - see java.util.Locale for more on valid codes. Note that even if org.teiid.comparableLobs is set, clob values will not be compared using the locale collator.	Not set by default, which means that Java's natural (UTF-16) string comparison will be used.
org.teiid.clientVdbLoadTimeoutMillis	The default amount of time a client (currently only local clients) will wait to make a connection to an active VDB before throwing an exception. Clients may override this setting via the loginTimeout connection property.	5 minutes
org.teiid.enDateNames	Set to true to use English month and day names for the system function dayName and monthName, rather than returning names from the Java default locale. Prior to 8.2 dayName and monthName always returned English names.	false
org.teiid.pushdownDefaultNullOrder	Set to true to mimic 8.1 and prior release behavior of pushing the Teiid's default null order of nulls low	false

	if the source has a different default null order and supports explicit null ordering.	
org.teiid.requireTeiidCollation	Set to true to force all sorts to be in Teiid’s collation (see org.teiid.collationLocale).	false
org.teiid.implicitMultiSourceJoin	Set to false to disable Teiid 8.2 and prior release behavior of implicitly partitioning joins between multi-source tables. When set to false and explicit predicate such as tbl1.source_name = tbl2.source_name is required to partition the results of the join.	true
org.teiid.maxStringLength	Sets the nominal maximum length of strings in Teiid - most operations in Teiid will truncate strings that are larger than this value. Setting this value can also adjust the max size of lob bytes held in memory. NOTE: sources may not appropriately handle string values that are larger than the source supports.	4000

Warning	Strings are always fully held in memory. Do not set this value too high as you may experience out of memory errors.

org.teiid.assumeMatchingCollation	If false and a translator does not specify a collationLocale, then a sort involving character data for a sort/merge join will not be pushed. Teiid defaults to the Java UCS-2 collation, which may not match the default collation for sources, particular tables, or columns. You may set the system property org.teiid.assumeMatchingCollation to true to restore the old default behavior or selectively update the translators to report a collationLocale matching org.teiid.collationLocale (UCS-2 if unset).	false
org.teiid.calendarTimestampDiff	Set to false to use the Teiid 8.2 and old computation of timestampdiff. note that: using the old behavior can result in differing results between pushed and non-pushed versions of timestampdiff for intervals greater than seconds as sources use date part and not approximate interval differences.	true
org.teiid.compactBufferFiles	Set to true to have Teiid keep the buffer files more compact (minimizing sparse regions).	false

org.teiid.maxMessageSize	The maximum size of messages in bytes that are allowed from clients. Increase only if clients routinely use large queries and/or non-lob bind values.	2097152
org.teiid.maxStreamingLobSize	The maximum size of lobs in bytes that are allowed to be streamed as part of the message from clients.	4294967296
org.teiid.defaultIndependentCardinality	The number of independent rows or less that can automatically trigger a dependent join. Increase when tables typically only have cardinality set and more dependent joins are desired.	10
org.teiid.checkPing	Can be set to false to disable ping checking for remote JDBC connections. Ping checking should only be disabled in specific circumstances, such as when using an external load balancer and not utilizing the Teiid default load balancing logic. Deprecated as of Teiid 10.2.	true
org.teiid.defaultNullOrder	Can be one of LOW, FIRST, HIGH, LAST. Sets the default null order for the Teiid engine. This will not be used for source ordering unless org.teiid.pushdownDefaultNullOrder is also set.	LOW
org.teiid.iso8601Week	Set to true to use ISO 8601 rules for week calculations regardless of the locale. When true the week function will require that week 1 of a year contains the year's first Thursday. Pushdown week values will be calculated as ISO regardless of this setting.	true
org.teiid.widenComparisonToString	Set to true to enable widening of values to string in comparisons, which was the default behavior prior to Teiid 9. For example with this setting as false timestamp_col < 'a' will produce an exception whereas when set to true it would effectively evaluate cast(timestamp_col as string) < `a'.	false
org.teiid.aggressiveJoinGrouping	Set to false to not aggressively group joins (typically allowed if there exists an explicit relationship) against the same source for pushdown and rely more upon a cost based ordering.	true
	Set to the desired size in bytes to limit the amount of buffer resources (heap and disk) consumed by a single session's tuple buffers and	

org.teiid.maxSessionBufferSizeEstimate	table structures. This is based upon the heap memory footprint estimate and may not correspond exactly to heap and especially to disk consumption. In general data in serialized from, whether on disk or in the fixed memory buffer, is between 3 and 8 times smaller than its heap representation which includes overhead such as additional object wrappers, lists, and less compact strings.	2^63 - 1
org.teiid.enforceSingleMaxBufferSizeEstimate	The system will determine an upper limit from all available memory for a single set of managed batches/pages - which could be a table, result set, or intermediate result - from all of the available buffer manager memory and disk. When this property is true an exception will be thrown when the limit is exceeded. When this property is false a TEIID31292 warning will be logged, which can be a good indicator of a query or environment that should be reviewed.	false
org.teiid.resultAnyPosition	Set to true to allow a RESULT parameter to appear at in position in a procedure parameter list.	false
org.teiid.requireUnqualifiedNames	Set to false to allow the pre-10.1 behavior of allowing qualified names in create to be used. For example 'create foreign table x.y …', rather than 'create foreign table "x.y" …'	true
org.teiid.useXMLxEscape	If _x escaping should be used for invalid characters in SQL/XML names. Set to false to use the older behavior of an _u escape.	true
org.teiid.tracingWithActiveSpanOnly	Set to false to always generate OpenTracing information even if no Span is active.	true
org.teiid.longRanks	Set to true to have the ranking functions RANK, DENSE_RANK, and ROW_NUMBER return long instead of integer.	false
org.teiid.relativeXPath	Set to true to have XPath PATH values beginning with / and // in XMLTABLE always be relative to the context item (the same behavior as Oracle). Set to false to have / and // PATH values to be evaluated from the root of the context item (the same behavior as PostgreSQL).	true

Security

Setting	Description	Default Value
org.teiid.allowAlter	If true alter and (sysdamin.setProperty) will be allowed at runtime to alter possibly ephemerally the metadata. If false those metadata alterations will not be allowed.	true
org.teiid.allowCreateTemporaryTablesByDefault	Set to true to use the pre-8.0 behavior of allowing any authenticated user to create temp tables without an explicit permission.	false
org.teiid.allowFunctionCallsByDefault	Set to true to use the pre-8.0 behavior of allowing any authenticated user to call any non-system function without an explicit permission.	false
org.teiid.hiddenMetadataResolvable	If true pg/JDBC objects under a hidden schema are still resolvable if fully qualified. If false objects under a hidden schema are never directly resolvable by an end user.	true
org.teiid.ignoreUnauthorizedAsterisk	If true unauthorized columns (as determined by data role checking) are not part of select all or qualified select all expansion. If false, the client may set the session variable ignore_unauthorized_asterisk to true to achieve the same behavior.	false
org.teiid.metadataRequiresPermission	If true metadata will only be visible in SYS/SYSADMIN tables if the user is permissioned in some way for the given object. If false the non-hidden schema metadata will be visible to any authenticated user.	true
org.teiid.ODBCRequireSecure	If true setting the SSL config to login or enabled will require clients to connect appropriately with either a GSS login or SSL respectively. Setting the property to false will allow client to use any authentication and no SSL (which was the behavior of the pg transport prior to 8.9 CR2).	true
org.teiid.sanitizeMessages	If true query related exception and warnings will have their messages replaced with just the Teiid code. Server side stacktraces will also be removed when sent to the client. This should be enabled if there is a concern about SQL or values being	false

	present in the exception/logs. If the log level is increased to debug for the relevant logger, then the sanitizeMessages setting will have no effect.	

PostgreSQL Compatibility

Note	These affect Teiid globally, and not just through the ODBC transport.

Setting	Description	Default Value
org.teiid.addPGMetadata	When set to false, the VDB will not include Postgresql based system metadata.	true
org.teiid.backslashDefaultMatchEscape	Set to true to use '\' as the default escape character for LIKE and SIMILAR TO predicates when no escape is specified. Otherwise Teiid assumes the SQL specification compliant behavior of treating each non-wildcard character as an exact match character.	false
org.teiid.honorDeclareFetchTxn	When false the wrapping begin/commit of a UseDeclareFetch cursor will be ignored as Teiid does not require a transaction.	false
org.teiid.pgVersion	Is the value that will be reported by the server_version function.	"PostgreSQL 8.2"

Client

System properties can also be set for client VMs. See Additional Socket Client Settings.

Teiid Management CLI

The WildFly CLI is a command line based administrative and monitoring tool for Teiid. Many snippets of CLI scripting are shown throughout the Admin Guide - especially around managing data sources. AdminAPI provides higher level methods that are often needed when interacting with Teiid. It is also useful to know the underlying CLI commands in many circumstances. The below is a series useful CLI commands for administering a Teiid Server. Please also refer to the AS documentation for more on interacting with the CLI - including how to navigate, list operations, etc.

Table of Contents

VDB Operations

```
deploy adminapi-test-vdb.xml
undeploy adminapi-test-vdb.xml

/subsystem=teiid:restart-vdb(vdb-name=AdminAPITestVDB, vdb-version=1, model-names=TestModel)

/subsystem=teiid:list-vdbs()
/subsystem=teiid:get-vdb(vdb-name=AdminAPITestVDB,vdb-version=1)
/subsystem=teiid:change-vdb-connection-type(vdb-name=AdminAPITestVDB, vdb-version=1,connection-type=ANY)

/subsystem=teiid:add-data-role(vdb-name=AdminAPITestVDB, vdb-version=1, data-role=TestDataRole, mapped-role=tes
t)
/subsystem=teiid:remove-data-role(vdb-name=AdminAPITestVDB, vdb-version=1, data-role=TestDataRole, mapped-role=
test)

/subsystem=teiid:read-attribute(name=async-thread-pool-max-thread-count)
/subsystem=teiid:write-attribute(name=async-thread-pool-max-thread-count,value=15)
```

Authentication Operations

```
/subsystem=teiid:read-attribute(name=authentication-security-domain)
/subsystem=teiid:write-attribute(name=authentication-security-domain,value=teiid-security)

/subsystem=teiid:read-attribute(name=authentication-max-sessions-allowed)
/subsystem=teiid:write-attribute(name=authentication-max-sessions-allowed,value=1000)

/subsystem=teiid:read-attribute(name=authentication-sessions-expiration-timelimit)
/subsystem=teiid:write-attribute(name=authentication-sessions-expiration-timelimit,value=0)

/subsystem=teiid:read-attribute(name=authentication-type)
/subsystem=teiid:write-attribute(name=authentication-type,value=USERPASSWORD)

/subsystem=teiid:read-attribute(name=authentication-trust-all-local)
/subsystem=teiid:write-attribute(name=authentication-trust-all-local,value=true)
```

Source Operations

```
/subsystem=teiid:add-source(vdb-name=AdminAPITestVDB, vdb-version=1, source-name=text-connector-test, translato
r-name=file, model-name=TestModel, ds-name=java:/test-file)
/subsystem=teiid:remove-source(vdb-name=AdminAPITestVDB, vdb-version=1, source-name=text-connector-test, model-
name=TestModel)
/subsystem=teiid:update-source(vdb-name=AdminAPITestVDB, vdb-version=1, source-name=text-connector-test, transl
ator-name=file, ds-name=java:/marketdata-file)
```

Translator Operations

```
/subsystem=teiid:list-translators()
/subsystem=teiid:get-translator(translator-name=file)
/subsystem=teiid:read-translator-properties(translator-name=file,type=OVERRIDE)
/subsystem=teiid:read-rar-description(rar-name=file)
```

Runtime Operations

```
/subsystem=teiid:workerpool-statistics()

/subsystem=teiid:cache-types()
/subsystem=teiid:clear-cache(cache-type=PREPARED_PLAN_CACHE)
/subsystem=teiid:clear-cache(cache-type=QUERY_SERVICE_RESULT_SET_CACHE)
/subsystem=teiid:clear-cache(cache-type=PREPARED_PLAN_CACHE, vdb-name=AdminAPITestVDB,vdb-version=1)
/subsystem=teiid:clear-cache(cache-type=QUERY_SERVICE_RESULT_SET_CACHE, vdb-name=AdminAPITestVDB,vdb-version=1)
/subsystem=teiid:cache-statistics(cache-type=PREPARED_PLAN_CACHE)
/subsystem=teiid:cache-statistics(cache-type=QUERY_SERVICE_RESULT_SET_CACHE)

/subsystem=teiid:engine-statistics()

/subsystem=teiid:list-sessions()
/subsystem=teiid:terminate-session(session=sessionid)

/subsystem=teiid:list-requests()
/subsystem=teiid:cancel-request(session=sessionId, execution-id=1)
/subsystem=teiid:list-requests-per-session(session=sessionId)
/subsystem=teiid:list-transactions()

/subsystem=teiid:mark-datasource-available(ds-name=java:/accounts-ds)

/subsystem=teiid:get-query-plan(session=sessionid,execution-id=1)
```

Diagnosing Issues

You may experience situations where you incur performance issues or unexpected results/exceptions. The rest of this chapter will focus on query planning and processing issues. Configuration or operational issues related to the container are typically more isolated and easier to resolve.

Table of Contents

General Diagnostic Process

- When there is an issue start by isolating a problem query as much as possible. OData, REST, and pg/ODBC access are layered on JDBC. If not accessing through JDBC, does the issue occur when using JDBC? If not, then the issue is at the transport layer rather than at the engine level. In whatever scenario the issue occurs, the particulars matter - what sources, if there is a transaction, load, etc.
- Don't make too many assumptions
  - For example memory consumption can be heavily dependent upon drivers, and a resulting out of memory issue may only be indirectly related to Teiid
- Start with the query plan - especially with performance issues
  - There may be simplifications or changes possible to views and procedures utilized by the user query.
  - Ensure that relevant costing metadata is set and/or that hints you have provided are being applied as expected.
- Utilize Logging
  - Planning issues may be understood with the debug plan
  - The command log
  - A full debug/trace level log can shed even more light – but it may not be easy to follow.
    - You can correlate what is happening by context, thread, session id, and request id.
- If no resolution is found, engage the community and utilize professional support

Query Plans

Once the problem has been isolated as much as possible, you should further examine the query plan. The only circumstance when this is not possible is when there are planning errors. In this case the logs, either full debug or just the debug plan, is still useful to then log an issue with the community or with support.

If you haven't done so already, you should start by familiarizing yourself with Federated Planning - especially the sections on the query plan.

The final processor plan is generally what is meant when referring to by "the query plan". The plan can be viewed in an XML or a plain text format.

You can also use Teiid Extensions, or SET/SHOW statements:

```
SET SHOWPLAN ON
SELECT ...
SHOW PLAN
```

or an Explain Statement:

```
EXPLAIN SELECT ...
```

Once you have the plan, you can:

- Double check that hints are taking effect
- Make sure things seem correct
  - Look first at all of the source queries on the access nodes. Generally a missing pushdown, such as predicate is easy to spot
  - Focus on problem source queries and their parent nodes if you already have execution times

It's also a good idea to validate query plans during the development and testing of a VDB. Also any engagement with the community or support will likely need the query plan as well.

If the plan is obtained from an executed query, then the plan will also show execution statistics. It is especially useful to see the stats when processing has finished and all rows have been fetched. While several stats are collected, it's most useful to see "Node Output Rows" and "Node Next Batch Process Time".

Example text form of a query plan showing stats:

```
ProjectNode
  + Relational Node ID:6
  + Output Columns:x (double)
  + Statistics:
    0: Node Output Rows: 6
    1: Node Next Batch Process Time: 2
    2: Node Cumulative Next Batch Process Time: 2
    3: Node Cumulative Process Time: 2
    4: Node Next Batch Calls: 8
    5: Node Blocks: 7
  + Cost Estimates:Estimated Node Cardinality: -1.0
  + Child 0:
    AccessNode
      + Relational Node ID:7
      + Output Columns
      + Statistics:
        0: Node Output Rows: 6
        1: Node Next Batch Process Time: 0
        2: Node Cumulative Next Batch Process Time: 0
        3: Node Cumulative Process Time: 0
        4: Node Next Batch Calls: 2
        5: Node Blocks: 1
...
```

In addition to the execution stats, note there are also cost estimates. The values for the cost estimates are derived from the statistic values set of each table/column about the row count, number of distinct values, number of null values, etc. Unlike systems that own the data, Teiid does not build histograms or other in-depth models of the data. Theses values are meant to be approximations with nominally distribution assumptions. The costing information from the metadata only matters for physical entities as we'll recompute the higher values in planning after merge virtual and other plan modifications. If you see that join is being implemented inefficiently, then first make sure reasonable costing values are being set on the tables involved. Statistics can be gathered for some sources at design time or deploy time. In environments that fluctuate rapidly, you may need to issue runtime costing updates via system procedures.

Note: if you cardinality values are unknown - shown as 'Node Cardinality: -1.0' in the plan - and no hints are used, then the optimizer will not assume that dependent join plans should be used.

Pushdown Inhibited

One of the most common issues that causes performance problems is when not enough of the plan is pushed to a given source leading to too many rows being fetched and/or too much processing in Teiid.

Pushdown problems fall into two categories:

- Something that cannot be pushed down. For example not all system functions are supported by each source. Formatting functions in particular are not broadly supported.
- A planning or other issue that prevents other constructs from being pushed down
  - Temp tables or materialization can inhibit pushdown when joining
  - Window functions and aggregation when not pushed can prevent further pushdown

If pushdown is inhibited then the construct will be missing from the access node issuing the source query, and will instead be be at a higher node:

```
<node name="SelectNode">...<property name="Criteria"><value>pm1.g1.e2 = 1</value>
  <node name="AccessNode">...<property name="Query"><value>SELECT pm1.g1.e1, pm1.g1.e2 FROM pm1.g1</value>
```

When pushdown is inhibited by the source, it should be easy to spot in the debug plan with log line similar to:

```
LOW Relational Planner SubqueryIn is not supported by source pm1 - e1 IN /*+ NO_UNNEST */ (SELECT e1 FROM pm2.g
1) was not pushed
```

Common Issues

Beyond pushdown being inhibited, other common issues are:

- Slight differences in Teiid/Pushdown results
  - for example Teiid produces a different for a given function than the source
- Source query form is not optimal or incorrect
- There is an unaccounted for type conversion
  - for example there is no char(n) type in Teiid
  - A cast may cause a source index not to be used
- Join Performance
  - Costing values not set leading to a non-performant plan.
  - Use hints if needed.
  - Teiid will replace outer with inner joins when possible, but just in case review outer join usage in the user query and view layers

XQuery

XQuery/XPath can be difficult to get correct when not assisted by tooling. Having an incorrect namespace for example could simply result in no results rather than exception.

With XMLQUERY/XMLTABLE each XPath/XQuery expression can have a large impact on performance. In particular descendant access '//' can be costly. Just accessing elements in the direct parentage is efficient though.

The larger the document being processed, the more careful you need to be to ensure that document projection and stream processing can be used. Streaming typically requires a simple context path - 'a/b/c'

Out of Memory

Out of memory errors can be difficult to track down. In almost all cases, it is best to determine the actual memory consumption utilizing a heap dump - which can be obtained using the vm HeapDumpOnOutOfMemoryError option or via a tool such as VisualVM. You may also simply increase the size of the heap, but that may simply delay the issue from reappearing.

Logging

The query plan alone does not provide a full accounting of processing. Some decisions are delayed until execution or can only be seen in the server logs:

- The ENAHANCED SORT JOIN node may execute can execute one of three different join strategies depending on the actually row counts found, this will not be seen unless the query plan is obtained at the end of execution.
- The effect of translation is not yet accounted for as the plan shows the engine form of the query
  - The full translation can be seen in with command logging at a trace level or with debug/trace logging in general.
- The query plan doesn’t show the execution stats of individual the source queries, which is shown in the command log
- The for full picture of execution down to all the batch fetches, you’ll just need the full server debug/trace log

Plan Debug Log

The logical plan optimization is represented by the planning debug log and is more useful to understand why planning decisions were made.

```
SET SHOWPLAN DEBUG
SELECT ...
SHOW PLAN
```

You will typically not need to use this level of detail to diagnose issues, but it is useful to provide the plan debug log to support when planning issues occur.

Migration Guide From Teiid 12.x to 13.x

Teiid strives to maintain consistency between all versions, but when necessary breaking configuration and VDB/sql changes are made - and then typically only for major releases.

You should consult the release notes for compatibility and configuration changes from each minor version that you are upgrading over. This guide expands upon the release notes included in the kit to cover changes since 12.x.

If possible you should make your migration to Teiid 13 by first using Teiid 12.2.x. Teiid 13 requires Java 8 and WildFly 17. See also 11 to 12 Migration Guide

Configuration Changes

The salesforce translators no longer support the ModelAuditFields execution property - the import property should be used instead.

Compatibility Changes

SET NAMESPACE

SET NAMESPACE should no longer be used. An exception will be thrown if the a custom namepsace or prefix is defined - only built-in namespaces/prefixes are allowed. Methods and constants related to namespaces have been removed. For now SYS.PROPERTIES will present built-in keys in both the old FQN format "{http…}key" and the new prefix format "teiid_…:key" so that existing SQL queries will work, but the legacy format will be removed in the next major release.

Security Changes

The target of GRANT/REVOKE statements will be validated against the metadata to ensure. Previous versions allowed the target to be any string.

The default data role enforcement will now check the strict hierarchy of a schema object, rather than every potential name part. In previous versions a table with a name containing "." such as "long.table.name" could have resulted in checks against permissions specified against the partial table names "long.table" and "long" as well. Now the will be a check only against the full table name, and then the schema.

The PolicyDecider was changed to reference the metadata objects rather than just strings. Any custom implementation will need to updated accordingly.

Kitting/Build Changes

The teiid-admin module/jar has been combined with teiid-api. Any references in custom development to teiid-admin should be replaces with teiid-api.

Migration Guide From Teiid 11.x to 12.x

Teiid strives to maintain consistency between all versions, but when necessary breaking configuration and VDB/sql changes are made - and then typically only for major releases.

You should consult the release notes for compatibility and configuration changes from each minor version that you are upgrading over. This guide expands upon the release notes included in the kit to cover changes since 11.x.

If possible you should make your migration to Teiid 12 by first using Teiid 11.2.x. Teiid 12 requires Java 8 and WildFly 14. See also 10 to 11 Migration Guide

Configuration Changes

System Properties

The default for org.teiid.longRanks changed to true. This is more inline with other platforms. You may switch it back to false for compatibility or make appropriate updates to your views and other sql that may expect integer values to be returned.

The default for org.teiid.enforceSingleMaxBufferSizeEstimate changed to false. Proactively limiting the size of a single operation does not match well to many Teiid usage scenarios, which could run just fine as long as enough disk was allocated. The default behavior will no selective kill sessions that are consuming the most amount of memory in response to running out of disk. It cannot be guaranteed that the current operation for with the disk running out will succeed however - in those circumstances it would be advisable to engage proactive limits.

Buffer Manager

The configuration property names for the buffer manager have been refined to greater consistency.

Old	New	Notes
buffer-service-processor-batch-size	buffer-manager-processor-batch-size	
buffer-service-inline-lobs	buffer-manager-inline-lobs instead	
buffer-service-max-processing-kb	buffer-manager-heap-max-processing-kb	
buffer-service-max-reserve-kb	buffer-manager-heap-max-reserve-mb	The unit change from kilobytes to megabytes
buffer-service-use-disk	buffer-manager-storage-enabled	Renamed to storage - which is both the fixed memory buffer and disk tiers below the buffer managed heap
buffer-service-max-storage-object-size	buffer-manager-storage-max-object-size-kb	The units changed from bytes to kilobytes
buffer-service-memory-buffer-space	buffer-manager-fixed-memory-space-mb	
buffer-service-memory-buffer-off-heap	buffer-manager-fixed-memory-off-heap	

buffer-service-max-file-size	buffer-manager-disk-max-file-size-mb	
buffer-service-max-buffer-space	buffer-manager-disk-max-space-mb	
buffer-service-max-open-files	buffer-manager-disk-max-open-files	
buffer-service-encrypt-files	buffer-manager-disk-encrypt-files	

This change introduced a new version of the WildFly xml configuration for Teiid. Older xml and cli are still compatible and are automatically converted to the new configuration. You should not mix the usage of new and old properties.

Compatibility Changes

The ability to specify a jgroups configuration file directly to Teiid Embedded has been removed. If you need Teiid Embedded to support clustering, please log an issue.

Kitting/Build Changes

AdminShell

The AdminShell has been removed from the build. AdminShell has releases between 10.x and 11.x are effectively identical. You may still use one of those versions if you wish to continue using AdminShell. Alternatively you may use the AdminAPI directly from Java or with the scripting language binding of your choice.

WildFly/JEE Restructuring

The maven coordinates for the full source, wildfly, and combined wildfly artifacts have changed. They were under org.teiid:teiid with classifiers that began with wildfly-. For example, instead of:

```
<groupId>org.teiid</groupId>
<artifactId>teiid</artifactId>
<classifier>wildfly-server</classifier>
<type>zip</type>
```

Use:

```
<groupId>org.teiid.wildfly</groupId>
<artifactId>teiid-wildfly</artifactId>
<classifier>server</classifier>
<type>zip</type>
```

Similarly all of the org.teiid:connector-xxx artifacts have moved to org.teiid.wildfly:connector-xxx. The Teiid embedded examples will be updated to reflect this change.

Dependencies on JEE have been pushed to non-core modules of Teiid. This should not affect anyone using a base Teiid distribution. However if you have done custom development it may affect you. The changes include:

- The `org.teiid.resource.spi` package was moved from the teiid-api jar to the org.teiid.wildfly:teiid-resource-spi jar. Poms will need to be updated accordingly. There is no change needed for jboss modules as the teiid-resource-spi artifact is already included in org.jboss.teiid.api.
- ResourceException has been replaced by TranslatorException on Teiid connection interfaces such as SalesforceConnection.
- The file translator resource adapter translator and connector logic were refactored to use a Teiid VirtualFile interface rather than directly expose both a Java File and VFS. If you were developing based upon FileConnection, please use VirtualFileConnection org.teiid.connectors:file-api instead.
- `org.teiid.translator.WSConnection` has been moved into org.teiid.connector:translator-ws `org.teiid.translator.ws.WSConnection`
- The arche-type version compatible with Teiid 12.0.0 has been bumped to 12.0.0.
- Teiid Embedded usage will need to include the org.teiid:cache-infinispan dependency, otherwise it will default to non-concurrent cache. If you are already setting the CacheFactory on the EmbeddedConfiguration, no action is needed. The EmbeddedConfiguration InfinispanConfigFile methods have been deprecated - in the future the user/platform will be fully responsible for wiring in the CacheFactory.

Migration Guide From Teiid 10.x to 11.x

Teiid strives to maintain consistency between all versions, but when necessary breaking configuration and VDB/sql changes are made - and then typically only for major releases.

You should consult the release notes for compatibility and configuration changes from each minor version that you are upgrading over. This guide expands upon the release notes included in the kit to cover changes since 10.x.

If possible you should make your migration to Teiid 11 by first using Teiid 10.3. Teiid 10.0 though 10.3 have the same JRE and WildFly requirements. Teiid 11 requires Java 8 and WildFly 11. See also 9 to 10 Migration Guide

Configuration Changes

The default max buffer space on disk for embedded, Spring, and Thorntail environments is 5 GB. The WildFly server environment default remains 50 GB, but needs to be specified in the configuration. If you are reusing the same configuration from Teiid 10 that has the default omitted, use the jboss cli to run:

/subsystem=teiid/:write-attribute(name=buffer-service-max-buffer-space, value=51200)

The authentication-allow-security-domain-qualifier property has been removed.

Compatibility Changes

Function model support has been completely removed as it had been deprecated in Teiid Designer for some time. Those models should be removed and the functions moved to other physical or virtual models.

The salesforce translator and resource adapter now provide access to the 34.0 version of the Salesforce API. You may need to re-import your salesforce source metadata to ensure compatibility.

Starting with 11.1 the Teiid client no longer supports the pluggable ServerDiscovery mechanism. The client will no longer support post-connection load-balancing nor a client side ping. If connecting to Teiid servers earlier than 10.2, then ping must be disabled on the server.

Session/user scoped materialized views are no longer supported. Please use a global temporary table instead.

Migration Guide From Teiid 9.x to 10.x

Teiid strives to maintain consistency between all versions, but when necessary breaking configuration and VDB/sql changes are made - and then typically only for major releases.

You should consult the release notes for compatibility and configuration changes from each minor version that you are upgrading over. This guide expands upon the release notes included in the kit to cover changes since 9.x.

If possible you should make your migration to Teiid 10 by first using Teiid 9.3. Teiid 9.1 though 9.3 have the same JRE and WildFly requirements. Teiid 10 requires Java 8 and WildFly 11. See also 8 to 9 Migration Guide

Configuration Changes

Teiid Embedded by default will not allow the usage of the ENV function. Use the EmbeddedConfiguration.setAllowEnvFunction to toggle this behavior.

Compatibility Changes

The FROM_UNIXTIME function now matches the behavior of HIVE/IMPALA. It accepts a long and returns a string, rather than a timestamp.

XML Document Model

The XML Document Model has been removed along with related client properties. Please consider migrating to OData or utilizing SQL/XML functions for constructing documents.

Kitting/Build Changes

The maven coordinates for Teiid artifacts has change from the org.jboss.teiid group to the org.teiid group. The artifacts are also published directly to maven central, rather than the JBoss nexus repository. This change was largely motivated by making the Teiid Spring integration less cumbersome. Note that this does not effect EAP/WildFly module names as those remain org.jboss.teiid.

Migration Guide From Teiid 8.x to 9.x

Teiid strives to maintain consistency between all versions, but when necessary breaking configuration and VDB/sql changes are made - and then typically only for major releases.

You should consult the release notes for compatibility and configuration changes from each minor version that you are upgrading over. This guide expands upon the release notes included in the kit to cover changes since 8.x.

If possible you should make your migration to Teiid 9 by first using Teiid 8.13. 8.13 is a non-feature transitional release that is effectively Teiid 8.12 running on WildFly 9.0.2. See also 8.13 Migration Guide

JRE Support

Teiid 9.1 uses WildFly 10.0.0. Both the server kit and usage of Teiid Embedded will require Java 1.8+. The client driver may still use a 1.6 runtime.

Teiid 9.0 uses WildFly 9.0.2. Both the server kit and usage of Teiid Embedded will require Java 1.7+. The client driver may still use a 1.6 runtime.

Configuration Changes

You will need to apply your Teiid and other configuration changes starting with a new base configuration for WildFly, such as the standalone-teiid.xml included in the kit. Note that 9999 port has been removed by default. Admin connections are expected to use either 9990 (http) or 9993 (https).

Security Related

There is now a single session service. Session service related properties, prefixed by authentication, are no longer specified per transport. Instead they now appear as a single sibling to the transports.

Old standalone.xml Configuration

```
<transport name="local"/>
<transport name="odata">
  <authentication security-domain="teiid-security"/>
</transport>
<transport name="jdbc" protocol="teiid" socket-binding="teiid-jdbc">
  <authentication security-domain="teiid-security"/>
</transport>
<transport name="odbc" protocol="pg" socket-binding="teiid-odbc">
  <authentication security-domain="teiid-security"/>
  <ssl mode="disabled"/>
</transport>
```

New standalone.xml Configuration

```
<authentication security-domain="teiid-security"/>

<transport name="local"/>
<transport name="odata"/>
<transport name="jdbc" protocol="teiid" socket-binding="teiid-jdbc"/>
<transport name="odbc" protocol="pg" socket-binding="teiid-odbc">
  <ssl mode="disabled"/>
</transport>
```

The default maximum number of sessions was increased to 10000 to accommodate for this change.

In addition there is a new property trust-all-local that defaults to true and allows unauthenticated access by local pass-through connections over the embedded transport - this was effectively the default behavior of 8.x and before when no security-domain was set on the embedded transport. You may choose to disallow that type of access by setting the property to false instead.

The authentication-security-domain property will only accept a single security domain, and will not interpret the value as a comma separated list. The default behavior has also changed for user names - they are longer allowed to be qualified by the security domain. Use the authentication-allow-security-domain-qualifier property to allow the old behavior of accepting user names that are security domain qualified.

RoleBasedCredentialMapIdentityLoginModule

The RoleBasedCredentialMapIdentityLoginModule class has been removed. Consider alternative login modules with roles assignments to restrict access to the VDB.

Local Transport

The embedded transport was renamed to local to avoid confusion with Teiid embedded.

Behavioral

widenComparisonToString

The resolver’s default behavior was to widen comparisons to string, but 9.0 now defaults org.teiid.widenComparisonToString to false. For example with this setting as false a comparison such as "timestamp_col < 'a'" will produce an exception whereas when set to true it would effectively evaluate "cast(timestamp_col as string) < 'a'". If you experience resolving errors when a vdb is deployed you should update the vdb if possible before reverting to the old resolving behavior.

reportAsViews

The JDBC client will report Teiid views in the metadata as table type VIEW rather than TABLE by default. Use the connection property reportAsViews=false to use pre-9.0 behavior.

Default Precision/Scale

If a column is specified with a precision of 0 or left as the default in DDL metadata it will be treated as having the nominal internal maximum value of 32767. This may cause the precision and scale to be reported differently, which may have been 2147483647 in some places or 20 in JDBC DatabaseMetaData.

Compatibility Changes

DDL Delimiters

Not using a semicolon delimiter between statements is deprecated and should only be relied on for backwards compatibility.

System Metadata

With data roles enabled system tables (SYS, SYSADMIN, and pg_catalog) will only expose tables, columns, procedures, etc. for which the user is entitled to access. A READ permission is expected for tables/columns, while an EXECUTE permission is expected for functions/procedures. All non-hidden schemas will still be visible though.

The OID columns has been removed. The UID column should be used instead or the corresponding pg_catalog table will contain an OID values.

Parent uid columns have been added to the SYS Tables, Procedures, KeyColumns, and Columns tables.

XML Document Model

The XML Document Model has been deprecated. Please consider migrating to OData or utilizing SQL/XML functions for constructing documents.

Kitting/Build Changes

Admin JAR

For 8.13 the entry point for creating remote admin connection, AdminFactory, was moved into the teiid-jboss-admin jar rather than being located in teiid-admin.

API Changes

The AuthorizationValidator and PolicyDecider interfaces had minor changes. AuthorizationValidator has an additional method to determine metadata filtering, and PolicyDecider had isTempAccessable corrected to isTempAccessible.

Semantic versioning required the change of the VDB version field from an integer to a string. This affected the following public classes:

VDB Session EventListener VDBImport ExecutionContext MetadataRepository

There are also duplicate/deprecated methods on:

EventDistributor Admin

Using the TranslatorProperty annotation without a setter now requires that readOnly=true be set on the annotation.

The JDBC DatabaseMetaData and CommandContext getUserName methods will now return just the base user name without the security domain.

Embedded Kit

The Embedded Kit has been removed. You should follow the Embedded Examples to use maven to pull the dependencies you need for your project.

There were extensive changes in dependency management for how the project is built. These changes allowed us to remove the need for resource adapter jars built with the lib classifier. If you need to reference these artifacts from maven, just omit the classifier.

Legacy Drivers

The drivers for JRE 1.4/1.5 systems have been discontinued. If you still need a client for those platforms, you should use the appropriate 8.x driver.

OData

The OData v2 war based upon odata4j has been removed. You should utilize the OData v4 war service instead.

The names of the wars have been changed to strip version information - this makes it easier to capture a deployment-overlay in the configuration such that it won’t be changed from one Teiid version to the next.

teiid-odata-odata2.war has become teiid-odata.war teiid-olingo-odata4.war has become teiid-olingo-odata4.war

To change properties in an web.xml file or add other files to the default odata war, you should use a deployment overlay instead.

Materialization

The semantic versioning change requires the materialization status tables to change their version column from an integer to string. Both the source and the source model will need to be updated with the column type change.

Caching Guide

Teiid provides several capabilities for caching data including:

1. Materialized views
2. ResultSet caching
3. Code table caching

These techniques can be used to significantly improve performance in many situations.

With the exception of external materialized views, the cached data is accessed through the BufferManager. In some cases the BufferManager setting can be adjusted to the particular memory constraints of your installation.

See the Cache Tuning for more on parameter tuning.

Results Caching

Teiid provides the capability to cache the results of specific user queries and virtual procedure calls. This caching technique can yield significant performance gains if users of the system submit the same queries or execute the same procedures often.

Support Summary

- Caching of user query results.
- Caching of virtual procedure results.
- Scoping of results is automatically determined to be VDB/user (replicated) or session level. The default logic will be influenced by every function evaluated, consider the DETERMINISM property on all source models/tables/procedures, and the Scope from the ExecutionContext or CacheDirective.
- Configurable number of cache entries and time to live.
- Administrative clearing.

User Interaction

User Query Cache

User query result set caching will cache result sets based on an exact match of the incoming SQL string and PreparedStatement parameter values if present. Caching only applies to SELECT, set query, and stored procedure execution statements; it does not apply to SELECT INTO statements, or INSERT, UPDATE, or DELETE statements.

End users or client applications explicitly state whether to use result set caching. This can be done by setting the JDBC ResultSetCacheMode execution property to true (default false)

```
Properties info = new Properties();
…
info.setProperty("ResultSetCacheMode", "true");
Connection conn = DriverManager.getConnection(url, info);
```

or by adding a Cache Hint to the query. Note that if either of these mechanisms are used, Teiid must also have result set caching enabled (the default is enabled).

The most basic form of the cache hint, `/*+ cache */`, is sufficient to inform the engine that the results of the non-update command should be cached.

PreparedStatement ResultSet Caching

```
PreparedStatement ps = connection.prepareStatement("/*+ cache */ select col from t where col2 = ?");
ps.setInt(1, 5);
ps.execute();
```

The results will be cached with the default ttl and use the SQL string and the parameter value as part of the cache key.

The pref_mem and ttl options of the cache hint may also be used for result set cache queries. If a cache hint is not specified, then the default time to live of the result set caching configuration will be used.

Advanced ResultSet Caching

```
/*+ cache(pref_mem ttl:60000) */ select col from t
```

In this example the memory preference has been enabled and the time to live is set to 60000 milliseconds or 1 minute. The ttl for an entry is actually treated as it's maximum age and the entry may be purged sooner if the maximum number of cache entries has been reached.

Note	Each query is re-checked for authorization using the current user's permissions, regardless of whether or not the results have been cached.

Procedure Result Cache

Similar to materialized views, cached virtual procedure results are used automatically when a matching set of parameter values is detected for the same procedure execution. Usage of the cached results may be bypassed when used with the OPTION NOCACHE clause. Usage is covered in Hints and Options.

Cached Virtual Procedure Definition

To indicate that a virtual procedure should be cached, it's definition should include a Cache Hint.

Procedure Caching

```
/*+ cache */
BEGIN
    ...
END
```

Results will be cached with the default ttl.

The `pref_mem` and `ttl` options of the cache hint may also be used for procedure caching.

Procedure results cache keys include the input parameter values. To prevent one procedure from filling the cache, at most 256 cache keys may be created per procedure per VDB.

A cached procedure will always produce all of its results prior to allowing those results to be consumed and placed in the cache. This differs from normal procedure execution which in some situations allows the returned results to be consumed in a streaming manner.

Cache Configuration

By default result set caching is enabled with 1024 maximum entries with a maximum entry age of 2 hours. There are actually 2 caches configured with these settings. One cache holds results that are specific to sessions and is local to each Teiid instance. The other cache holds VDB scoped results and can be replicated. See the teiid subsystem configuration for tuning. The user may also override the default maximum entry age via the Cache Hint.

Result set caching is not limited to memory. There is no explicit limit on the size of the results that can be cached. Cached results are primarily stored in the *BufferManager* and are subject to it's configuration - including the restriction of maximum buffer space.

While the result data is not held in memory, cache keys - including parameter values - may be held in memory. Thus the cache should not be given an unlimited maximum size.

Result set cache entries can be invalidated by data change events. The *max-staleness* setting determines how long an entry will remain in the case after one of the tables that contributed to the results has been changed.

See the Developer’s Guide for further customization.

Extension Metadata

You can use the extension metadata property

```
{http://www.teiid.org/ext/relational/2012}data-ttl
```

as a schema/model property or on a source table to indicate a default TTL. A negative value means no TTL, 0 means do not cache, and a positive number indicates the time to live in milliseconds. If no TTL is specified on the table, then the schema will be checked. The TTL for the cache entry will be taken as the least positive value among all TTLs. Thus setting this value as a model property can quickly disable any caching against a particular source.

For example, setting the property in the vdb:

```
CREATE DATABASE vdbname;
USE DATABASE vdbname;
...
CREATE SCHEMA PM1 SERVER connector OPTIONS ("teiid_rel:data-ttl" 0);
...
```

As an XML VDB:

```
<vdb name="vdbname" version="1">
    <model name="Customers">
        <property name="teiid_rel:data-ttl" value="0"/>
        ...
```

Cache Administration

The result set cache can be cleared through the AdminAPI using the `clearCache` method. The expected cache key is "QUERY_SERVICE_RESULT_SET_CACHE".

Clearing the ResultSet Cache in AdminAPI

```
admin.clearCache("QUERY_SERVICE_RESULT_SET_CACHE")
```

See the AdminAPI for more on using the AdminAPI.

Limitations

- XML, BLOB, CLOB, JSON, Geometry, and OBJECT type cannot be used as part of the cache key for prepared statement of procedure cache keys.
- The exact SQL string, including the cache hint if present, must match the cached entry for the results to be reused. This allows cache usage to skip parsing and resolving for faster responses.
- Result set caching is transactional by default using the NON_XA transaction mode. If you want full XA support, then change the configuration to use NON_DURABLE_XA.
- Clearing the results cache clears all cache entries for all VDBs.

Materialized Views

Teiid supports materialized views. Materialized views are just like other views, but their transformations are pre-computed and stored just like a regular table. When queries are issued against the views through the Teiid Server, the cached results are used. This saves the cost of accessing all the underlying data sources and re-computing the view transformations each time a query is executed.

Materialized views are appropriate when the underlying data does not change rapidly, or when it is acceptable to retrieve data that is "stale" within some period of time, or when it is preferred for end-user queries to access staged data rather than placing additional query load on operational sources.

Support Summary

- Caching of relational table or view records (pre-computing all transformations)
- Model-based definition of virtual groups to cache
- User ability to override use of materialized view cache for specific queries through Hints and Options

Approach

The overall strategy toward materialization should be to work on the integration model first, then optimize as needed from the top down.

Result set caching, ideally hint driven, should be used if there lots of repeated user queries. If result set caching is insufficient, then move onto internal materialization for views that are closest to consumers (minimally or not layered) that are introducing performance issues. Keep in mind that the use of materialization inlines access to the materialization table rather than the view so scenarios that integrate on top of the materialization may suffer if they were relying on pushing/optimizing the work of the view with surrounding constructs.

Based upon the limitations of internal materialization, then switch to external materialization as needed.

Materialized View Definition

Materialized views are defined in by setting the materialized property on a table or view in a virtual (view) relational model. Setting this property's value to true (the default is false) allows the data generated for this virtual table to be treated as a materialized view.

Important	It is important to ensure that all key/index information is present as these will be used by the materialization process to enhance the performance of the materialized table.

The target materialized table may also be set in the properties. If the value is left blank, the default, then internal materialization will be used. Otherwise for external materialization, the value should reference the fully qualified name of a table (or possibly view) with the same columns as the materialized view. For most basic scenarios the simplicity of internal materialization makes it the more appealing option.

Reasons to use external materialization

- The cached data needs to be fully durable. Internal materialization does not survive a cluster restart.

- Full control is needed of loading and refresh. Internal materialization does offer several system supported methods for refreshing, but does not give full access to the materialized table.
- Control is needed over the materialized table definition. Internal materialization does support Indexes, but they cannot be directly controlled. Constraints or other database features cannot be added to internal materialization tables.
- The data volume is large. Internal materialization (and temp tables in general) have memory overhead for each page. A rough guideline is that there can be 100 million rows in all materialized tables across all VDBs for every gigabyte of heap.

Important	Materialized view tables default to the VDB scope. By default if a materialized view definition directly or transitively contains a non-deterministic function call, such as random or hasRole, the resulting table will contain only the initially evaluated values. In most instances you should consider nesting a materialized view without the deterministic results that is joined with relevant non-deterministic values in a parent view.

Important	Nearly all of the materialization related properties must be set at the time the vdb is loaded and are not monitored for changes. Removal of properties at runtime, such as the status table, will result in exceptions.

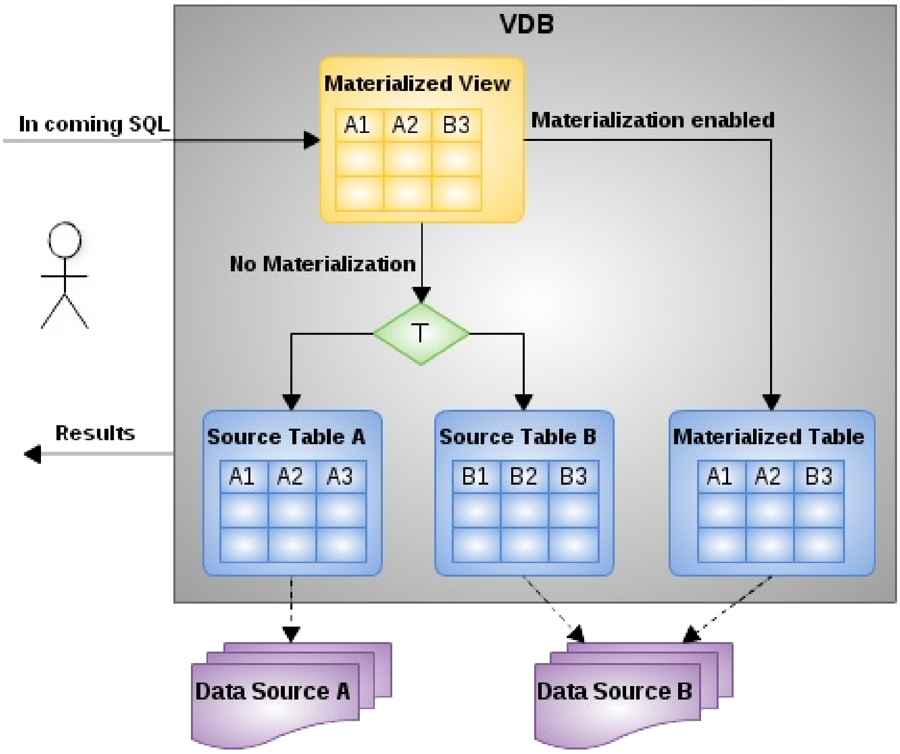
External Materialization

This document will explain what Teiid External Materialization is and how to use it.

Table of Contents

What is it ?

In Teiid, a view is a virtual table based on the computing(loading/transforming/federating) of a complex SQL statement across heterogeneous data sources. Teiid external materialization process can cache the View data to an external data source systems on a periodic basis. When a user issues queries against this View, the request will be redirected to this external data source system where cached results will be returned, rather than re-computing results from source systems. Materialization can prove to be time and resource saving if your View transformation is complex and/or access to the source systems is constrained.



Materialized View - Materialized view is just like other views, with additional options in View Options, to enable pre-computing and caching data to an external data source system.

Materialized Table - Materialized table represents the target table for the materialized View, has the same structure as the materialized view, but exists on the external data source system.

MatView Status Table - Each materialized view has a reference to 'Status' table, this used to save the Materialized views' refresh status. This table typically exists on the same physical source with the `Materialized Table`.

An external materialized view gives the administrator full control over the loading and refresh strategies. Refer to Materialization Management for details.

External Materialized Data Source Systems

The following are the types of data sources that have been tested to work in the external materialization process:

RDBMS Systems

- RDBMS - a relational database should work. Example databases; Oracle, Postgresql, MySQL, MS SqlServer, SAP Hana, etc.

If the database supports a transactional rename operation, you can use the default load strategy that uses a staging table and rely on renaming the staging table to the live table in the after load script.

Note	TEIID-4294 raises that not every database supports a transactional rename, either as separate or a block of statements. If this is the case you should consider using a LOADNUMBER column, or a custom load strategy that maintains only a single table.

Infinispan

- Infinispan - for in-memory caching of results. see the Infinispan Translator.

View Options

The following View properties are extension properties that used in the management of the Materialized View.

Property Name	Description	Optional	D
MATERIALIZED	Set the value to 'TRUE' for the View to be materialized.	false	n/a
MATERIALIZED_TABLE	Defines the name of target table, this also hints the materialization is using external materialization. Omitting this property and setting the MATERIALIZED property true, invokes internal materialization.	false	n/a
UPDATABLE	Allow updating Materialized View via DML updates	true	false
teiid_rel:ALLOW_MATVIEW_MANAGEMENT	Allow Teiid based automatic management of load/refresh strategies of View.	true	false
teiid_rel:MATVIEW_STATUS_TABLE	Fully qualified Status Table Name to manage the load/refresh of the materialized view. See below for table structure and DDL for it.	false	n/a

teiid_rel:MATVIEW_LOAD_SCRIPT - DEPRECATED	command to run for loading of the cache. Use of this property is deprecated in favor of using the "MATVIEW_LOADNUMBER_COLUMN" property.	true	will deter base trans
teiid_rel:MATERIALIZED_STAGE_TABLE - DEPRECATED	When MATVIEW_LOAD_SCRIPT property not defined, Teiid loads the cache contents into this table. Required when MATVIEW_LOAD_SCRIPT not defined. Use of this property is deprecated in favor using the "MATVIEW_LOADNUMBER_COLUMN" property.	true	n/a
teiid_rel:MATVIEW_LOADNUMBER_COLUMN	Name of column in the `MATERIALIZED_TABLE` that can hold status information about load/refresh load process. The column type MUST be long, and typically named as "LoadNumber".	false	NON
teiid_rel:MATVIEW_BEFORE_LOAD_SCRIPT	DDL/DML command to run before the actual load of the cache	true	Whe defir scrip run
teiid_rel:MATVIEW_AFTER_LOAD_SCRIPT	DDL/DML command to run after the actual load of the cache. teiid_rel:MATVIEW_STAGE_TABLE to MATVIEW table	true	Whe defir scrip run
teiid_rel:MATVIEW_SHARE_SCOPE	Allowed values are {IMPORTED, FULL}, which define if the cached contents are shared among different VDB versions and different imported VDBs and parent VDB.	true	IMP
teiid_rel:ON_VDB_START_SCRIPT	DDL/DML command to run start of vdb	true	n/a
teiid_rel:ON_VDB_DROP_SCRIPT	DDL/DML command to run at VDB un-deploy; typically used for cleaning the cache/status tables. DO NOT use this script to delete the contents of Status table, when cache scope settings are configured for {FULL} scope, **if** another version of the VDB is still active. Deletion of this information will reload the materialization table.	true	n/a
teiid_rel:MATVIEW_ONERROR_ACTION	Action to be taken when mat view contents are requested but cache is invalid. Allowed values are (THROW_EXCEPTION = throws an exception, IGNORE = ignores the warning and supplied invalidated data, WAIT = waits until the data is refreshed and valid then provides the updated data)	true	WAI
teiid_rel:MATVIEW_TTL	time to live in milliseconds. Provide property or cache hint on view transformation - property takes precedence.	true	2^63 milli effec table refre

			will a sin initia
teiid_rel:MATVIEW_WRITE_THROUGH	When true Teiid will perform both the underlying update and the corresponding update against the materialization target for an insert/update/delete issued against the view.	true	false
teiid_rel:MATVIEW_MAX_STALENESS_PCT	This property defines the percentage max of staleness allowed before a refresh to the View is invoked. Any double value 0 to 100 is valid value. The *StateCount* column on Status table is used to keep track of the number of updates, and this value is checked against Cardinality column to calculate the amount of variance. The availability of this property, supercedes the MATVIEW_TTL property interms of when a refresh job triggred to update the contents of the view.	true	n/a
teiid_rel:MATVIEW_POLLING_QUERY	This property defines a query that must return a single timestamp value. If the value is greater than the last update time of the materialization table, it will be reloaded.	true	n/a
teiid_rel:MATVIEW_POLLING_INTERVAL	This property defines the polling interval, in milliseconds, used with the polling query and STALENESS_PCT based refreshes.	true	6000

Tip	for scripts that need more than one statement executed, use a procedure block BEGIN statement; statement; … END

Important	When a vdb is imported into another vdb, materializied views are automatically shared across these vdbs. The teiid_rel:MATVIEW_SHARE_SCOPE property must be set to 'IMPORTED' or 'FULL' on importing VDB’s materialized views to enable sharing across the both vdbs. The below table shows an example of how this property works

For example: Table A is in VDB X.1 and Table C in VDB Y.1 Table A & B in VDB X.2 and imports Y.1 then depending on scope setting the system will cache sharing will work as

Scope	X.1	Y.1	X.2
IMPORTED	A-own copy	C-Shared w/X.2	A-own copy,B-own copy,C-Shared from Y.1
FULL	A-Shared with/X.*	C-Shared w/X.2	A-Shared with/ X,B-Shared w/X,C-Shared from/Y.1

An example View definition with View Options

```
CREATE VIEW Person (
  id varchar,
  name varchar,
  dob date,
  PRIMARY KEY (id)
```

```
) OPTIONS (
  MATERIALIZED 'TRUE',
  UPDATABLE 'TRUE',
  MATERIALIZED_TABLE 'materialized.PersonCached',
  "teiid_rel:MATVIEW_TTL" 20000,
  "teiid_rel:ALLOW_MATVIEW_MANAGEMENT" 'true',
  "teiid_rel:MATVIEW_LOADNUMBER_COLUMN" 'LoadNumber',
  "teiid_rel:MATVIEW_STATUS_TABLE" 'materialized.status'
)
AS
  SELECT p.id, p.name, p.dob FROM Source.Person AS p;
```

Materialization Management

When designing Views, you can define additional metadata and extension properties(refer to above section) on the views to control the loading and refreshing of external materialization cache. This option provides a limited, but a powerful way to manage the materialization views. Below we will list steps need to take to configure a View to be materialized.

1. Creation of Status Table

To manage and report the loading and refreshing activity of materialization of the view, a **Materialized Table** and **Status Table** need be be defined in one of the source models in the VDB. Create these tables on the physical database, before you deploy the VDB.

The below defines the DDL for creating the Status table.

```
CREATE TABLE status
(
  VDBName varchar(50) not null,
  VDBVersion varchar(50) not null,
  SchemaName varchar(50) not null,
  Name varchar(256) not null,
  TargetSchemaName varchar(50),
  TargetName varchar(256) not null,
  Valid boolean not null,
  LoadState varchar(25) not null,
  Cardinality long,
  Updated timestamp not null,
  LoadNumber long not null,
  NodeName varchar(25) not null,
  StaleCount long,
  PRIMARY KEY (VDBName, VDBVersion, SchemaName, Name)
);
```

Appendix-1: DDL for creating MatView Status Table contains a series of verified schemas against different RDBMS sources. These can be modified to suit your database, please make sure the names and data types match exactly.

Warning	Some databases, such as MySQL with the InnoDB backend, may not allow a large primary key such as the one for the status table. If you experience this, you should consider making the field sizes shorter (such as the table name), using a different database to hold the status, or using a smaller index (for example just over vdbname and vdbversion).

Description Status table:

Column Name	Description

VDBName	Name of VDB
VDBVersion	Version of VDB
SchemaName	View’s Schema
TargetSchemaName	Schema name of materialization Table
TargetName	Name of materialization Table
Valid	true when view materialization contents are valid; false otherwise
LoadState	Status of the View; LOADING, LOADED, FAILED_LOAD. During the materialization load, this status is set to LOADING, depending upon the success or failure either LOADED or FAILED_LOAD is set.
Cardinality	Number of rows loaded
Updated	Time stamp when the last update occurred on the materialization contents
LoadNumber	Counter to keep track of number of updates to the materialization contents
NodeName	Node name, which updated the materialization contents last
StaleCount	Number updates counted against View, based on source table changes when using LAZY-SNAPSHOT strategy.

2. Creation of View and Materialized Table

Define the View and its transformation a VDB’s model/schema. Then provide the extension properties on the View as defined in View Options

Set the `MATERIALIZED` to 'TRUE' and the `MATERIALIZED_TABLE` point to a target table is necessary for external materialization, `UPDATABLE` is optional, set it to 'TRUE' if want the external materialized view be updatable, this must be set to true, if you want to issue incremental eager updates to the view. Define the TTL to define the load/refresh semantics.

In an another **PHYSICAL** model in the VDB (where the Status table defined), define the Materialized table, where the **Materialized Table** should have the same structure as View it is representing, with additional "LoadNumber" column with "long" data type.

Once a View, which is defined with the above properties, is deployed, the following sequence of events will take place:

Tip	Example VDB based on DDL is defined below for reference.

Materialization Table Loading

Upon deployment of the VDB to the Teiid server, SYSADMIN.loadMatView used to perform a complete refresh of materialized table, this procedure reads the extension properties defined from View Options to customize the load. The following describes the sequence of events that occur inside this procedure

1. Inserts/updates an entry in `teiid_rel:MATVIEW_STATUS_TABLE` , which indicates that the cache is being loaded.
2. Executes `teiid_rel:MATVIEW_BEFORE_LOAD_SCRIPT` if defined.
3. Runs a query to load the cache contents. This makes use of View's transformation to load the contents.
4. Executes `teiid_rel:MATVIEW_AFTER_LOAD_SCRIPT` if defined.
5. Updates `teiid_rel:MATVIEW_STATUS_TABLE` entry to set materialized view status status to "LOADED" and valid. If failure happens it will be marked as such.

Tip	The start/stop scripts are not cluster aware - that is they will run on each cluster member as the VDB is deployed. When deploying into a clustered environment, the scripts should be written in such a way as to be cluster safe.

Once the first load of the materialized view, the update/refresh of the this View is controlled by the extension property "MATVIEW_TTL" or "MATVIEW_MAX_STALENESS_PCT". Currently there are three different refresh types allowed

Refresh Type: TTL Based SNAPSHOT

Based on the MATVIEW_TTL extension property defined on View, when the time configured is elapsed from the time of finish of loading the View, the whole view is reloaded automatically if the "ALLOW_MATVIEW_MANAGEMENT" property is set to true. If the contents are externally managed additional properties are required. Note, that "MATVIEW_MAX_STALENESS_PCT" is not provided in this case.

Refresh Type: LAZY SNAPSHOT

This is similar to TTL Based SNAPSHOT, but differs as to what triggers the reload of the view. Every source table update(s) is captured in the Status table's StaleCount column as single updated event, and when this updated count reaches or exceeds the defined "MATVIEW_MAX_STALENESS_PCT" value, then a full refresh is triggered. The values of StaleCount/Cardinality are used to calculate the percent of variance to invoke the trigger for refresh. Also note this refresh type only applies when view is materialized to external sources. `SYSADMIN.updateStaleCount` procedure is used to increment the StaleCount counter. When integrated with CDC technologies like Debezium (new feature coming..) this procedure is called automatically.

Refresh Type: EAGER

When a view refresh type is defined as "EAGER", the very first time the contents if the materialized view are loaded similar to that of other types using the `SYSADMIN.loadMatView` procedure upon the deployment of the VDB. However, once the contents are loaded, SYSADMIN.updateMatView can be used to perform a eager incremental update based on any criteria provided. If you know that certain data points in the source system were changed after last full refresh of the materialized view, you can call this procedure with a criteria based on the view that cover those changed values, and this procedure will update only those affected rows in the materialized table instead of doing full snapshot update. This can save lot of time and resources and also keeps your view materialization cache upto date with source system changes.

Note: This script is not invoked automatically by Teiid, as the source update events may be occurring outside of Teiid. This procedure needs to be invoked by user, when he/she knows that there is change in the source systems. When CDC technologies like Debezium is used (new feature coming..), this procedure can be automatically invoked to keep the the View contents fresh.

Appendix-1: DDL for creating MatView Status Table

h2

```
CREATE TABLE status
(
  VDBName varchar(50) not null,
  VDBVersion varchar(50) not null,
```

```
  SchemaName varchar(50) not null,
  Name varchar(256) not null,
  TargetSchemaName varchar(50),
  TargetName varchar(256) not null,
  Valid boolean not null,
  LoadState varchar(25) not null,
  Cardinality long,
  Updated timestamp not null,
  LoadNumber long not null,
  NodeName varchar(25) not null,
  StaleCount long,
  PRIMARY KEY (VDBName, VDBVersion, SchemaName, Name)
);
```

MariaDB

```
CREATE TABLE status
(
  VDBName varchar(50) not null,
  VDBVersion varchar(50) not null,
  SchemaName varchar(50) not null,
  Name varchar(256) not null,
  TargetSchemaName varchar(50),
  TargetName varchar(256) not null,
  Valid boolean not null,
  LoadState varchar(25) not null,
  Cardinality bigint,
  Updated timestamp not null,
  LoadNumber bigint not null,
  NodeName varchar(25) not null,
  StaleCount bigint,
  PRIMARY KEY (VDBName, VDBVersion, SchemaName, Name)
);
```

Appendix-2: Example VDB with External Materialized View Options

The below VDB defines three models, one "Source" model that defines your source database where your business data is in, "ViewModel" defines a "Person" view which is derived from subset of the data from your table in the "Source" model’s table(s). Note that view table also marked with few extension properties to allow external materialization. The "materialized" model defines a source database model, where it has a table with exact table structure as the ViewModel’s materialized view with additional column called "LoadNumber". Note the "materialized table also contains the "status" table. Both these tables must be created manually on the source database before VDB is deployed to the server. The example below uses TTL_SNAPSHOT based refresh.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<vdb name="example" version="1">
    <model name="Source">
        <source name="source" translator-name="h2" connection-jndi-name="java:/my-ds" />
    </model>

    <model name="ViewModel" type="VIRTUAL">
        <metadata type="DDL"><![CDATA[
          CREATE VIEW Person (
            id varchar,
            name varchar,
            dob date,
            PRIMARY KEY (id)
          ) OPTIONS (
            MATERIALIZED 'TRUE', UPDATABLE 'TRUE',
            MATERIALIZED_TABLE 'materialized.PersonCached',
            "teiid_rel:MATVIEW_TTL" 20000,
```

```
            "teiid_rel:ALLOW_MATVIEW_MANAGEMENT" 'true',
            "teiid_rel:MATVIEW_LOADNUMBER_COLUMN" 'LoadNumber',
            "teiid_rel:MATVIEW_STATUS_TABLE" 'materialized.status'
          )
          AS
            SELECT p.id, p.name, p.dob FROM Source.Person AS p;
        ]]>
        </metadata>
    </model>

    <model name="materialized" type="PHYSICAL">
        <source name="matview" translator-name="h2" connection-jndi-name="java:/matview-ds" />
        <metadata type="DDL"><![CDATA[
          CREATE VIEW PersonCached (
            id varchar,
            name varchar,
            dob date,
            LoadNumber long,
            PRIMARY KEY (id)
          );
          CREATE TABLE status (
            VDBName varchar(50) not null,
            VDBVersion varchar(50) not null,
            SchemaName varchar(50) not null,
            Name varchar(256) not null,
            TargetSchemaName varchar(50),
            TargetName varchar(256) not null,
            Valid boolean not null,
            LoadState varchar(25) not null,
            Cardinality long,
            Updated timestamp not null,
            LoadNumber long not null,
            NodeName varchar(25) not null,
            StaleCount long,
            PRIMARY KEY (VDBName, VDBVersion, SchemaName, Name)
          ) OPTIONS (UPDATABLE true);
        ]]>
        </metadata>
    </model>
</vdb>
```

Internal Materialization

Internal materialization creates Teiid temporary tables to hold the materialized table. While these tables are not fully durable, they perform well in most circumstances and the data is present at each Teiid instance which removes the single point of failure and network overhead of an external database. Internal materialization also provides built-in facilities for refreshing and monitoring.

See Memory Limitations regarding size limitations.

Table of Contents

View Options

The materialized option must be set for the view to be materialized. The Cache Hint, when used in the context of an internal materialized view transformation query, provides the ability to fine tune the materialized table. The caching options are also settable via extension metadata:

Property Name	Description	Optional	Default
materialized	Set for the view to be materialized	false	true
UPDATABLE	Allow updating Materialized View via DML `UPDATE`	true	false
teiid_rel:ALLOW_MATVIEW_MANAGEMENT	Allow Teiid based management of the ttl and initial load rather than the implicit behavior.	true	false
teiid_rel:MATVIEW_PREFER_MEMEORY	Same as the pref_mem cache hint option.	true	false
teiid_rel:MATVIEW_TTL	Trigger a Scheduled ExecutorService which execute refreshMatView repeatedly with a specified time to live	true	null
teiid_rel:MATVIEW_UPDATABLE	Allow updating Materialized View via refreshMatView, refreshMatViewRow, refreshMatViewRows	true	false.

teiid_rel:MATVIEW_SCOPE	Same as the scope cache hint option.	true	VDB
teiid_rel:MATVIEW_WRITE_THROUGH	When true Teiid will perform both the underlying update and the corresponding update against the materialization target for an insert/update/delete issued against the view.	true	false
teiid_rel:MATVIEW_POLLING_QUERY	This property defines a query that must return a single timestamp value. If the value is greater than the last update time of the materialization table, it will be reloaded.	true	n/a
teiid_rel:MATVIEW_POLLING_INTERVAL	This property defines the polling interval, in milliseconds, used with the polling query.	true	60000

The pref_mem option also applies to internal materialized views. Internal table index pages already have a memory preference, so the perf_mem option indicates that the data pages should prefer memory as well.

All internal materialized view refresh and updates happen atomically. Internal materialized views support READ_COMMITTED (used also for READ_UNCOMMITED) and SERIALIZABLE (used also for REPEATABLE_READ) transaction isolation levels.

A sample VDB defining an internal materialization

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

    <model name="pg">
        <source name="pg" translator-name="postgresql" connection-jndi-name="java:/sakila-ds"/>
    </model>

    <model name="sakila" type="VIRTUAL">
    <metadata type="DDL"><![CDATA[
        CREATE VIEW actor (
           actor_id integer,
           first_name varchar(45) NOT NULL,
           last_name varchar(45) NOT NULL,
           last_update timestamp NOT NULL
        ) OPTIONS (materialized true,
               UPDATABLE 'TRUE',
               "teiid_rel:MATVIEW_TTL" 120000,
               "teiid_rel:MATVIEW_PREFER_MEMORY" 'true',
               "teiid_rel:ALLOW_MATVIEW_MANAGEMENT" 'true',
               "teiid_rel:MATVIEW_UPDATABLE" 'true',
               "teiid_rel:MATVIEW_SCOPE" 'vdb')
            AS SELECT actor_id, first_name, last_name, last_update from pg."public".actor;
     ]]>
    </metadata>
    </model>
</vdb>
```

Loading And Refreshing

An internal materialized view table is initially in an invalid state (there is no data).

- If `teiid_rel:ALLOW_MATVIEW_MANAGEMENT` is specified as true, then the initial load will occur on vdb startup.
- If `teiid_rel:ALLOW_MATVIEW_MANAGEMENT` is not specified or false, then the load of the materialization table will occur on implicit on the first query that accesses the table.

When a refresh happens while the materialization table is invalid all other queries against the materialized view will block until the load completes.

Using System Procedure

In some situations administrators may wish to better control when the cache is loaded with a call to `SYSADMIN.refreshMatView`. The initial load may itself trigger the initial load of dependent materialized views. After the initial load user queries against the materialized view table will only block if it is in an invalid state. The valid state may also be controlled through the `SYSADMIN.refreshMatView` procedure.

Invalidating Refresh

```
CALL SYSADMIN.refreshMatView(viewname=>'schema.matview', invalidate=>true)
```

matview will be refreshed and user queries will block until the refresh is complete (or fails).

While the initial load may trigger a transitive loading of dependent materialized views, subsequent refreshes performed with `refreshMatView` will use dependent materialized view tables if they exist. Only one load may occur at a time. If a load is already in progress when the `SYSADMIN.refreshMatView` procedure is called, it will return -1 immediately rather than preempting the current load.

Using TTL Snapshot Refresh

The Cache Hint or extension properties may be used to automatically trigger a full snapshot refresh after a specified time to live (ttl). The behavior is different depending on whether the materialization is managed or non-managed.

For non-managed views the ttl starts from the time the table is finished loading and the refresh will be initiated after the ttl has expired on a view access.

For managed views the ttl is a fixed interval and refreshes will be triggered regardless of view usage.

In either case the refresh is equivalent to `CALL SYSADMIN.refreshMatView('view name', *)`, where the invalidation behavior * is determined by the vdb property lazy-invalidate. By default ttl refreshes are invalidating, which will cause other user queries to block while loading. That is once the ttl has expired, the next access will be required to refresh the materialized table in a blocking manner. If you would rather that the ttl is enforced lazily, such that the current contents are not replaced until the refresh completes, set the vdb property lazy-invalidate=true.

Auto-refresh Transformation Query*

```
/*+ cache(ttl:3600000) */ select t.col, t1.col from t, t1 where t.id = t1.id
```

The resulting materialized view will be reloaded every hour (3600000 milliseconds).

TTL Snapshot Refresh Limitations

- The automatic ttl refresh may not be suitable for complex loading scenarios as nested materialized views will be used by the refresh query.
- The non-managed ttl refresh is performed lazily, that is it is only trigger by using the table after the ttl has expired. For infrequently used tables with long load times, this means that data may be used well past the intended ttl.

Updatable

In advanced use-cases the cache hint may also be used to mark an internal materialized view as updatable. An updatable internal materialized view may use the `SYSADMIN.refreshMatViewRow` procedure to update a single row in the materialized table. If the source row exists, the materialized view table row will be updated. If the source row does not exist, the correpsonding materialized row will be deleted. To be updatable the materialized view must have a single column primary key. Composite keys are not yet supported by `SYSADMIN.refreshMatViewRow`. Transformation Query:

```
/*+ cache(updatable) */ select t.col, t1.col from t, t1 where t.id = t1.id
```

Update SQL:

```
CALL SYSADMIN.refreshMatViewRow(viewname=>'schema.matview', key=>5)
```

Given that the schema.matview defines an integer column col as its primary key, the update will check the live source(s) for the row values.

The update query will not use dependent materialized view tables, so care should be taken to ensure that getting a single row from this transformation query performs well. See the Reference Guide for information on controlling dependent joins, which may be applicable to increasing the performance of retrieving a single row. The refresh query does use nested caches, so this refresh method should be used with caution.

When the updatable option is not specified, accessing the materialized view table is more efficient because modifications do not need to be considered. Therefore, only specify the updatable option if row based incremental updates are needed. Even when performing row updates, full snapshot refreshes may be needed to ensure consistency.

The `EventDistributor` also exposes the updateMatViewRow as a lower level API for Programmatic Control - care should be taken when using this update method.

Indexes

Internal materialized view tables will automatically create a unique index for each unique constraint and a non-unique index for each index defined on the materialized view. The primary key (if it exists) of the view will automatically be part of a clustered index.

The secondary indexes are always created as ordered trees - bitmap or hash indexes are not supported. Teiid's metadata for indexes is currently limited. We are not currently able to capture additional information, sort direction, additional columns to cover, etc. You may workaround some of these limitations though.

- Function based index are supported, but can only be specified through DDL metadata. If you are not using DDL metadata, consider adding another column to the view that projects the function expression, then place an index on that new column. Queries to the view will need to be modified as appropriate though to make use of the new column/index.
- If additional covered columns are needed, they may simply be added to the index columns. This however is only applicable to comparable types. Adding additional columns will increase the amount of space used by the index, but may allow its usage to result in higher performance when only the covered columns are used and the main table is not consulted.

Clustering Considerations

Each member in a cluster maintains its own copy of each materialized table and associated indexes. An attempt is made to ensure each member receives the same full refresh events as the others. Full consistency for updatable materialized views however is not guaranteed. Periodic full refreshes of updatable materialized view tables helps ensure consistency among members.

Code Table Caching

Teiid provides a short cut to creating an internal materialized view table via the *lookup* function.

The *lookup* function provides a way to accelerate getting a value out of a table when a key value is provided. The function automatically caches all of the key/return pairs for the referenced table. This caching is performed on demand, but will proactively load the results to other members in a cluster. Subsequent lookups against the same table using the same key and return columns will use the cached information.

This caching solution is appropriate for integration of "reference data" with transactional or operational data. Reference data is usually static and small data sets that are used frequently. Examples are ISO country codes, state codes, and different types of financial instrument identifiers.

Usage

This caching mechanism is automatically invoked when the lookup scalar function is used. The lookup function returns a scalar value, so it may be used anywhere an expression is expected. Each time this function is called with a unique combination of referenced table, return column, and key column (the first 3 arguments to the function).

See the Lookup Function in the Reference Guide for more information on use of the *lookup* function.

Country Code Lookup

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

Limitations

- The use of the lookup function automatically performs caching; there is no option to use the lookup function and not perform caching.
- No mechanism is provided to refresh code tables
- Only a single key/return column is cached - values will not be session/user specific.

Materialized View Alternative

The *lookup* function is a shortcut to create an internal materialized view with an appropriate primary key. In many situations, it may be better to directly create the analogous materialized view rather than to use a code table.

Country Code Lookup Against A Mat View

```
SELECT (SELECT CountryCode From MatISOCountryCodes WHERE CountryName = tbl.CountryName) as cc FROM tbl
```

Here MatISOCountryCodes is a view selecting from ISOCountryCodes that has been marked as materialized and has a primary key and index on CountryName. The scalar subquery will use the index to lookup the country code for each country name in tbl.

Reasons to use a materialized view:

- More control of the possible return columns. Code tables will create a materialized view for each key/value pair. If there are multiple return columns it would be better to have a single materialized view.
- Proper materialized views have built-in system procedure/table support.

- More control via the cache hint.
- The ability to use OPTION NOCACHE.
- There is almost no performance difference.

Steps to create a materialized view:

1. Create a view selecting the appropriate columns from the desired table. In general, this view may have an arbitrarily complicated transformation query.
2. Designate the appropriate column(s) as the primary key. Additional indexes can be added if needed.
3. Set the materialized property to true.
4. Add a cache hint to the transformation query. To mimic the behavior of the implicit internal materialized view created by the lookup function, use the Hints and Options `/*+ cache(pref_mem) */` to indicate that the table data pages should prefer to remain in memory.

Just as with the lookup function, the materialized view table will be created on first use and reused subsequently. See the Materialized Views for more.

Translator Results Caching

Translators can contribute cache entries into the result set cache via the use of the `CacheDirective` object. The resulting cache entries behave just as if they were created by a user query. See the Translator Caching API for more on this feature.

Cache Hint

A query cache hint can be used to:

- Indicate that a user query is eligible for result set caching and set the cache entry memory preference, time to live, etc.
- Set the materialized view memory preference, time to live, or updatablity.
- Indicate that a virtual procedure should be cachable and set the cache entry memory preference, time to live, etc.

```
/*+ cache[([pref_mem] [ttl:n] [updatable] [scope:session|user|vdb])]*/ sql ...
```

- The cache hint should appear at the beginning of the SQL. It can be appear as any one of the leading comments. It will not have any affect on INSERT/UPDATE/DELETE statements or INSTEAD OF TRIGGERS.
- *pref_mem*- if present indicates that the cached results should prefer to remain in memory. The results may still be paged out based upon memory pressure.

Note	Care should be taken to not over use the pref_mem option. The memory preference is implemented with Java soft references. While soft references are effective at preventing out of memory conditions. Too much memory held by soft references can limit the effective working memory. Consult your JVM options for clearing soft references if you need to tune their behavior.

- *ttl:n*- if present n indicates the time to live value in milliseconds. The default value for result set caching is the default expiration for the corresponding Infinispan cache. There is no default time to live for materialized views.
- *updatable*- if present indicates that the cached results can be updated. This defaults to false for materialized views and to true for result set cache entries.
- *scope*- There are three different cache scopes: session - cached only for current session, user - cached for any session by the current user, vdb - cached for any user connected to the same vdb. For cached queries the presence of the scope overrides the computed scope. Materialized views can only be the vdb scope.

The *pref_mem*, *ttl*, *updatable*, and *scope* values for a materialized view may also be set via extension properties on the view - using the teiid_rel namespace with MATVIEW_PREFER_MEMORY, MATVIEW_TTL, MATVIEW_UPDATABLE, and MATVIEW_SCOPE respectively. If both are present, the use of an extension property supersedes the usage of the cache hint.

Limitations

The form of the query hint must be matched exactly for the hint to have affect. For a user query if the hint is not specified correctly, e.g. `/*+ cach(pref_mem) */`, it will not be used by the engine nor will there be an informational log. It is currently recommended that you verify (see Client Developers Guide) in your testing that the user command in the query plan has retained the proper hint.

OPTION NOCACHE

Individual queries may override the use of cached results by specifying `OPTION NOCACHE` on the query. 0 or more fully qualified view or procedure names may be specified to exclude using their cached results. If no names are specified, cached results will not be used transitively.

Full NOCACHE

```
SELECT * from vg1, vg2, vg3 WHERE … OPTION NOCACHE
```

No cached results will be used at all.

Specific NOCACHE

```
SELECT * from vg1, vg2, vg3 WHERE … OPTION NOCACHE vg1, vg3
```

Only the vg1 and vg3 caches will be skipped, vg2 or any cached results nested under vg1 and vg3 will be used.

`OPTION NOCACHE` may be specified in procedure or view definitions. In that way, transformations can specify to always use real-time data obtained directly from sources.

Programmatic Control

Teiid exposes a bean that implements the *org.teiid.events.EventDistributor* interface. It can be looked up in JNDI under the name *teiid/event-distributor-factory*. The `EventDistributor` exposes methods like *dataModification* (which affects result set caching) or *updateMatViewRow* (which affects internal materialization) to alert the Teiid engine that the underlying source data has been modified. These operations, which work cluster wide will invalidate the cache entries appropriately and reload the new cache contents.

Note	**Change Data Capture** - If your source system has any built-in change data capture facilities that can scrape logs, install triggers, etc. to capture data change events, they can captured and can be propagated to Teiid engine through a pojo bean/MDB/Session Bean deployed in WildFly engine.

The below shows a code example as how user can use `EventDistributor` interface in their own code that is deployed in the same WildFly VM using a Pojo/MDB/Session Bean. Consult WildFly documents deploying as bean as they out of scope for this document.

EventDistributor Code Example

```
public class ChangeDataCapture {

    public void invalidate() {
        InitialContext ic = new InitialContext();
        EventDistributor ed = ((EventDistributorFactory)ic.lookup("teiid/event-distributor-factory")).getEventD
istributor();

        // this below line indicates that Customer table in the "model-name" schema has been changed.
        // this result in cache reload.
        ed.dataModification("vdb-name", "version", "model-name", "Customer");
    }
}
```

Note	**Updating Costing information** - The *EventDistributor* interface also exposes many methods that can be used to update the costing information on your source models for optimized query planning. Note that these values volatile and will be lost during a cluster re-start, as there is no repository to persist.

Developing clients for Teiid

This guide intended for developers that are trying to write 3rd party applications that interact with Teiid. You can find information about connection mechanisms, extensions to the JDBC API, ODBC, SSL and so forth.

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.

JDBC compatibility

Teiid provides a robust JDBC driver that implements most of the JDBC API according to the latest specification and compatible Java version. Most tooling designed to work with JDBC should work seamlessly with the Teiid driver. When in doubt, see Incompatible JDBC Methods for functionality that has yet to be implemented.

If your needs go beyond JDBC, Teiid has also provided JDBC Extensions for asynch handling, federation, and other features.

Generated Keys

Teiid can return generated keys for JDBC sources and from Teiid temp tables with SERIAL primary key columns. However the current implementation will return only the last set of keys generated and will return the key results directly from the source - no view projection of other intermediate handling is performed. For most scenarios (single source inserts) this handling is sufficient. A custom solution may need to be developed if you are using a FOR EACH ROW instead of trigger to process your inserts and target multiple tables that each return generated keys. It is possible to develop a UDF that also manipulates the returned generated keys - see the `org.teiid.CommandContext` methods dealing with generated keys for more.

Note	You cannot use Generated Keys when the JDBC Batched updates is used to insert the values into the source table.

Connecting to a Teiid Server

The Teiid JDBC API provides Java Database Connectivity (JDBC) access to a Virtual Database (VDB) deployed on Teiid. The Teiid JDBC API is compatible with the JDBC 4.0 specification; however, it is not compatible with some methods. You cannot use some advanced features, such as updatable result sets or SQL3 data types.

Java client applications connecting to a Teiid Server will need to use at least the Java 1.8 JDK. Earlier versions of Java are not compatible. You may attempt to use a client driver from earlier Teiid versions that were compatible with the target JRE.

Support for Teiid clients and servers older than version 8 has been dropped from Teiid 10.2 and later.

Before you can connect to the Teiid Server using the Teiid JDBC API, please do following tasks.

1. Install the Teiid Server. See the "Admin Guide" for instructions.
2. Build a Virtual Database (VDB). Check the "Reference Guide" for instructions on how to build a VDB. If you do not know what VDB is, then start with this document.
3. Deploy the VDB into Teiid Server. Check Administrator’s Guide for instructions.
4. Start the Teiid Server (WildFly), if it is not already running.

After you deploy the virtual database, client applications can connect to it and issue SQL queries against it using the JDBC API. If you are new to JDBC, refer to the Java documentation about JDBC. Teiid ships with `teiid-VERSION_NUMBER-jdbc.jar` that is available from the Teiid.io downloads.

You can also obtain the Teiid JDBC from the Maven repository at https://oss.sonatype.org/content/repositories/releases/ using the coordinates:

```
<dependency>
  <groupId>org.teiid</groupId>
  <artifactId>teiid</artifactId>
  <classifier>jdbc</classifier>
  <version>$versionNumber</version>
</dependency>
```

where `versionNumber` is the version of the most recent Teiid release.

Important classes in the client JAR:

- `org.teiid.jdbc.TeiidDriver` - allows JDBC connections using the DriverManager class.
- `org.teiid.jdbc.TeiidDatasource` - allows JDBC connections using the DataSource XADataSource class. You should use this class to create managed or XA connections.

Once you have established a connection with the Teiid Server, you can use standard JDBC API classes to interrogate metadata and execute queries.

OpenTracing compatibility

OpenTracing is optional for the client driver. For remote connections to propagate the span the driver must have the appropriate OpenTracing jars in its classpath. This can be done via a maven dependency:

```
<dependency>
    <groupId>io.opentracing</groupId>
    <artifactId>opentracing-util</artifactId>
    <version>${version.opentracing}</version>
```

```
</dependency>
```

where `version.opentracing` is defined in the project integration bom.

Alternately, you can manually include the `opentracing-util` , `opentracing-api` , and `opentracing-noop` jar files as needed by the tooling or other environment where the Teiid client jar is utilized.

OpenTracing support in the client and server requires that the respective runtimes have an appropriate tracing client installed and available via the GlobalTracer.

Driver Connection

Use **org.teiid.jdbc.TeiidDriver** as the driver class.

Use the following URL format for JDBC connections:

```
jdbc:teiid:<vdb-name>[@mm[s]://<host>:<port>][;prop-name=prop-value]*
```

Note	The JDBC client will have both JRE and server compatibility considerations. Unless otherwise stated a client jar will typically be forward and backwards compatible with one major version of the server. You should attempt to keep the client up-to-date though as fixes and features are made on to the client.

URL Components

1. <vdb-name> - Name of the VDB you are connecting to. Optionally VDB name can also contain version information inside it. For example: "myvdb.2", this is equivalent to supplying the "version=2" connection property defined below. However, use of vdb name in this format and the "version" property at the same time is not allowed.
2. mm - defines Teiid JDBC protocol, mms defines a secure channel (see SSL Client Connections for more)
3. <host> - defines the server where the Teiid Server is installed. If you are using IPv6 binding address as the host name, place it in square brackets. ex:[::1]
4. <port> - defines the port on which the Teiid Server is listening for incoming JDBC connections.
5. [prop-name=prop-value] - additionally you can supply any number of name value pairs separated by semi-colon [;]. All compatible URL properties are defined in the connection properties section. Property values should be URL encoded if they contain reserved characters, e.g. ('?', '=', ';', etc.)

Note	host and port may be a comma separated list to specify multiple hosts.

Local Connections

To make a in-VM connection, omit the protocol and host/port: jdbc:teiid:vdb-name;props

For local WildFly deployments it's preferred to configure the DataSource as an in-VM rather than socket based connection.

URL Connection Properties

The following table shows all the connection properties that you can use with Teiid JDBC Driver URL connection string, or on the Teiid JDBC Data Source class.

Table 1. **Connection Properties**

Property Name	Type	Description
`ApplicationName`	`String`	Name of the client application; allows the administrator to identi connections
`FetchSize`	`int`	Size of the resultset; The default size if 500. ⇐0 indicates that th should be used.
`partialResultsMode`	`boolean`	Enable/disable partial results mode. Default false. See the Partial Mode section.

autoCommitTxn	String	Only applies only when "autoCommit" is set to "true". This dete executed command needs to be transactionally wrapped inside th engine to maintain the data integrity. • ON - Always wrap command in distributed transaction • OFF - Never wrap command in distributed transaction • DETECT (default)- If the executed command is spanning m source it automatically uses distributed transaction. Transac more information.
disableLocalTxn	boolean	If "true", the autoCommit setting, commit and rollback will be ig local transactions. Default false.
user	String	User name
Password	String	Credential for user
ansiQuotedIdentifiers	boolean	Sets the parsing behavior for double quoted entries in SQL. The parses doubled quoted entries as identifiers. If set to false, then d values that are valid string literals will be parsed as string literals
version	integer	Version number of the VDB
resultSetCacheMode	boolean	ResultSet caching is turned on/off. Default false.
autoFailover	boolean	If true, will automatically select a new server instance after a co exception. Default false. This is typically not needed when conn managed, as the connection can be purged from the pool. If true mode, connections will reconnect to a newer VDB of the same n
SHOWPLAN	String	(typically not set as a connection property) Can be ON, OFF,DE • ON returns the query plan along with the results • DEBUG additionally prints the query planner debug inform log and returns it with the results. Both the plan and the log through JDBC API extensions. • Default OFF.
NoExec	String	(typically not set as a connection property) Can be ON, OFF; O query execution, but parsing and planning will still occur. Defau
PassthroughAuthentication	boolean	Only applies to "local" connections. When this option is set to "t Teiid looks for already authenticated security context on the calli one found it uses that users credentials to create session. Teiid al that the same user is using this connection during the life of the c it finds a different security context on the calling thread, it switc identity on the connection, if the new user is also eligible to log i otherwise connection fails to execute.
useCallingThread	boolean	Only applies to "local" connections. When this option is set to "t default), then the calling thread will be used to process the query then an engine thread will be used.
QueryTimeout	integer	Default query timeout in seconds. Must be >= 0. 0 indicates no ti be overriden by `Statement.setQueryTimeout` . Default 0.

`useJDBC4ColumnNameAndLabelSemantics`	`boolean`	A change was made in JDBC4 to return unaliased column names ResultSetMetadata column name. Prior to this, if a column alias was returned as the column name. Setting this property to false w backwards compatibility with JDBC3 and earlier. Defaults to tru
`jaasName`	`String`	JAAS configuration name. Only applies when configuring a GSS authentication. Defaults to Teiid. See the Security Guide for conf required for GSS.
`kerberosServicePrincipleName`	`String`	Kerberos authenticated principle name. Only applies when confi authentication. See the Security Guide for configuration required
`encryptRequest`	`boolean`	Only applies to non-SSL socket connections. When "true" the re message and any associate payload will be encrypted using the c cryptor. Default false.
`disableResultSetFetchSize`	`boolean`	In some situations tooling may choose undesirable fetch sizes fo results. Set to true to disable honoring ResultSet.setFetchSize. D
`loginTimeout`	`integer`	The login timeout in seconds. Must be >= 0. 0 indicates no speci but other timeouts may apply. If a connection cannot be created i approximately the the timeout value an exception will be thrown 0 does not mean that the login will wait indefinitely. Typically if vdb cannot be found, the login will fail at that time. Local conne specify a vdb version however can wait by default for up to the t in the property org.teiid.clientVdbLoadTimeoutMillis[org.teiid.clientVdbLoadT
`reportAsViews`	`boolean`	If DatabaseMetaData will report Teiid views as a VIEW table typ then Teiid views will be reported as a TABLE. Default true.

DataSource Connection

To use a data source based connection, use `org.teiid.jdbc.TeiidDataSource` as the data source class. The `TeiidDataSource` is also an XADatasource. Teiid DataSource class is also Serializable, so it possible for it to be used with JNDI naming services.

Teiid is compatible with the XA protocol, XA transactions will be extended to Teiid sources that support XA.

All the properties (except for version, which is known on TeiidDataSource as DatabaseVersion) defined in the Driver Connection#URL Connection Properties have corresponding "set" methods on the `org.teiid.jdbc.TeiidDataSource`. Properties that are assumed from the URL string have additional "set" methods, which are described in the following table.

Table 1. **Datasource Properties**

Property Name	Type	Description
`DatabaseName`	`String`	The name of a virtual database (VDB) deployed to Teiid. Optionally Database name can also contain "DatabaseVersion" information inside it. For example: "myvdb.2", this is equivalent to supplying the "DatabaseVersion" property set to value of 2. However, use of Database name in this format and use of DatabaseVersion property at the same time is not allowed.
`ServerName`	`String`	Server hostname where the Teiid runtime installed. If you are using IPv6 binding address as the host name, place it in square brackets. ex:[::1]
`AlternateServers`	`String`	Optional delimited list of host:port entries. See the Using Multiple Hosts for more information. If you are using IPv6 binding address as the host name, place them in square brackets. ex:[::1]
`AdditionalProperties`	`String`	Optional setting of properties that has the same format as the property string in a connection URL.
`PortNumber`	`integer`	Port number on which the Server process is listening on.
`secure`	`boolean`	Secure connection. Flag to indicate to use SSL (mms) based connection between client and server
`DatabaseVersion`	`integer`	VDB version
`DataSourceName`	`String`	Name given to this data source

Note	**Additional Properties** - All the properties from URL Connection Properties can be used on DataSource using the *AdditionalProperties* setter method if the corresponding setter method is not already available. For example, you can add *"useCallingThread"* property as `<xa-datasource-property name="AdditionalProperties">useCallingThread=false</xa-datasource-property>`

Standalone Application

To use either Driver or DataSource based connections, add the client JAR to your Java client application's classpath. See the simple client example in the kit for a full Java sample of the following.

Driver Connection

Sample Code:

```
public class TeiidClient {
    public Connection getConnection(String user, String password) throws Exception {
        String url = "jdbc:teiid:myVDB@mm://localhost:31000;ApplicationName=myApp";
        return DriverManager.getConnection(url, user, password);
    }
}
```

Datasource Connection

Sample Code:

```
public class TeiidClient {
    public Connection getConnection(String user, String password) throws Exception {
        TeiidDataSource ds = new TeiidDataSource();
        ds.setUser(user);
        ds.setPassword(password);
        ds.setServerName("localhost");
        ds.setPortNumber(31000);
        ds.setDatabaseName("myVDB");
        return ds.getConnection();
    }
}
```

WildFly DataSource

Teiid can be configured as a JDBC data source in a WildFly Server to be accessed from JNDI or injected into your JEE applications. Deploying Teiid as data source in WildFly is exactly same as deploying any other RDBMS resources like Oracle or DB2.

Defining as data source is not limited to WildFly, you can also deploy as data source in Glassfish, Tomcat, Websphere, Weblogic etc servers, however their configuration files are different than WildFly. Consult the respective documentation of the environment in which you are deploying.

A special case exists if the Teiid instance you are connecting to is in the same VM as the WildFly instance. If that matches you deployment, then follow the Local JDBC Connection instructions

Installation Steps

1. If you are working with an AS instance that already has Teiid installed then required module / jar files are already installed. If the AS instance does not have Teiid installed, then you should create a module for the client jar. Under the path module/org/jboss/teiid/client add the client jar and a module.xml defined as:

   Sample Teiid Client Module

   ```
   <module xmlns="urn:jboss:module:1.1" name="org.jboss.teiid.client">
       <resources>
           <resource-root path="teiid-{version}-jdbc.jar"/>
       </resources>

       <dependencies>
           <module name="javax.api"/>
           <module name="javax.transaction.api"/>
       </dependencies>
   </module>
   ```

Note	Prior to Teiid 8.12.3 a module dependency on sun.jdk was also required.

2. Use the CLI or edit the `standalone-teiid.xml` or `domain-teiid.xml` file and add a datasource into the "datasources" subsystem.

Based on the type of deployment (XA, driver, or local), the contents of this will be different. See the following sections for more. The data source will then be accessible through the JNDI name specified in the below configuration.

DataSource Connection

Make sure you know the correct DatabaseName, ServerName, Port number and credentials that are specific to your deployment environment.

Sample XADataSource in the WildFly using the Teiid DataSource class org.teiid.jdbc.TeiidDataSource

```
<datasources>
        <xa-datasource jndi-name="java:/teiidDS" pool-name="teiidDS" enabled="true" use-java-context="true" use
-ccm="true">

            <xa-datasource-property name="PortNumber">31000</xa-datasource-property>
            <xa-datasource-property name="DatabaseName">{db-name}</xa-datasource-property>
            <xa-datasource-property name="ServerName">{host}</xa-datasource-property>

            <driver>teiid</driver>
            <xa-pool>
                <min-pool-size>10</min-pool-size>
```

```
                <max-pool-size>20</max-pool-size>
                <is-same-rm-override>true</is-same-rm-override>
                <prefill>false</prefill>
                <use-strict-min>false</use-strict-min>
                <flush-strategy>FailingConnectionOnly</flush-strategy>
                <no-tx-separate-pools/>
            </xa-pool>
            <security>
                <user-name>{user}</user-name>
                <password>{password}</password>
            </security>
        </xa-datasource>
        <drivers>
            <driver name="teiid" module="org.jboss.teiid.client">
                <driver-class>org.teiid.jdbc.TeiidDriver</driver-class>
                <xa-datasource-class>org.teiid.jdbc.TeiidDataSource</xa-datasource-class>
            </driver>
        </drivers>
    </datasources>
```

Driver based connection

You can also use Teiid's JDBC driver class `org.teiid.jdbc.TeiidDriver` to create a data source

```
<datasources>
        <datasource jndi-name="java:/teiidDS" pool-name="teiidDS">
            <connection-url>jdbc:teiid:{vdb}@mm://{host}:31000</connection-url>
            <driver>teiid</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>{password}</password>
            </security>
        </datasource>
        <drivers>
            <driver name="teiid" module="org.jboss.teiid.client">
                <driver-class>org.teiid.jdbc.TeiidDriver</driver-class>
                <xa-datasource-class>org.teiid.jdbc.TeiidDataSource</xa-datasource-class>
            </driver>
        </drivers>
    </datasources>
```

Local JDBC Connection

If you are deploying your client application on the same WildFly instance as the Teiid runtime is installed, then you will want to configure the connection to by-pass making a socket based JDBC connection. By using a slightly different data source configuration to make a "local" connection, the JDBC API will lookup a local Teiid runtime in the same VM.

Warning	Since DataSources start before Teiid VDBs are deployed, leave the min pool size of 0 for local connections. Otherwise errors may occur on the startup of the Teiid DataSource. Also note that local connections specifying a VDB version will wait for their VDB to be loaded before allowing a connection. See loginTimeout and the org.teiid.clientVdbLoadTimeoutMillis system property.

Warning	Do not include any additional copy of Teiid jars in the application classload that is utilizing the local connection. Even if the exact same version of the client jar is included in your application classloader, you will fail to connect to the local connection with a class cast exception.

Note	By default local connections use their calling thread to perform processing operations rather than using an engine thread while the calling thread is blocked. To disable this behavior set the connection property *useCallingThreads=false*. The default is true, and is recommended in transactional queries.

Local data source

```
<datasources>
        <datasource jndi-name="java:/teiidDS" pool-name="teiidDS">
            <connection-url>jdbc:teiid:{vdb}</connection-url>
            <driver>teiid-local</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>{password}</password>
            </security>
        </datasource>
        <drivers>
            <driver name="teiid-local" module="org.jboss.teiid">
                <driver-class>org.teiid.jdbc.TeiidDriver</driver-class>
                <xa-datasource-class>org.teiid.jdbc.TeiidDataSource</xa-datasource-class>
            </driver>
        </drivers>
    </datasources>
```

This is essentially the same as the XA configuration, but *"ServerName"* and *"PortNumber"* are not specified. Local connections have additional features such as using PassthroughAuthentication

Using Multiple Hosts

A group of Teiid Servers in the same WildFly cluster may be connected using failover and load-balancing features.

External HA / Load Balancers

You may choose to use an external tcp load balancer, such as haproxy. The Teiid driver/DataSource should then typically be configured to just use the single host/port of your load balancer.

Even if you configure the load balancer to redirect when there is a failed host, that will not maintain the Teiid session state. If you wish to keep the connection alive, then use the *autoFailover* feature discussed below. Otherwise the other Teiid Client Features are not necessary when using an external load balancer.

Teiid Client Features

To enable theses features in their simplest form, the client needs to specify multiple host name and port number combinations on the URL connection string.

Example URL connection string

```
jdbc:teiid:<vdb-name>@mm://host1:31000,host1:31001,host2:31000;version=2
```

If you are using a DataSource to connect to Teiid Server, use the "AlternateServers" property/method to define the failover servers. The format is also a comma separated list of host:port combinations.

The client will randomly pick one the Teiid server from the list and establish a session with that server. If that server cannot be contacted, then a connection will be attempted to each of the remaining servers in random order. This allows for both connection time fail-over and random server selection load balancing.

Fail Over

Post connection fail over will be used if the *autoFailover* connection property on JDBC URL is set to true. Post connection failover works by sending a ping, at most every second, to test the connection prior to use. If the ping fails, a new instance will be selected prior to the operation being attempted.

This is not true "transparent application failover" as the client will not restart the transaction/query/recreate session scoped temp tables, etc. So this feature should be used with caution.

Client SSL Settings

The following sections define the properties required for each SSL mode. Note that when connecting to Teiid Server with SSL enabled, you *MUST* use the *"mms"* protocol, instead of "mm" in the JDBC connection URL, for example

Note	Anonymous SSL mode is not provided for some JREs, see the Teiid Server Transport Security for alternatives.

```
jdbc:teiid:<myVdb>@mms://<host>:<port>
```

There are two different sets of properties that a client can configure to enable 1-way or 2-way SSL.

See also the Teiid Server Transport Security chapter if you are responsible for configuring the server as well.

Option 1: Java SSL properties

These are standard Java defined system properties to configure the SSL under any JVM, Teiid is not unique in its use of SSL. Provide the following system properties to the client VM process.

1-way SSL

```
-Djavax.net.ssl.trustStore=<dir>/server.truststore (required)
-Djavax.net.ssl.trustStorePassword=<password> (optional)
-Djavax.net.ssl.keyStoreType (optional)
```

2-way SSL

```
-Djavax.net.ssl.keyStore=<dir>/client.keystore (required)
-Djavax.net.ssl.keyStrorePassword=<password> (optional)
-Djavax.net.ssl.trustStore=<dir>/server.truststore (required)
-Djavax.net.ssl.trustStorePassword=<password> (optioanl)
-Djavax.net.ssl.keyStroreType=<keystore type> (optional)
```

Option 2: Teiid Specific Properties

Use this option when the above "javax" based properties are already in use by the host process. For example if your client application is a Tomcat process that is configured for https protocol and the above Java based properties are already in use, and importing Teiid-specific certificate keys into those https certificate keystores is not allowed.

In this scenario, a different set of Teiid-specific SSL properties can be set as system properties or defined inside the a "teiid-client-settings.properties" file. A sample "teiid-client-settings.properties" file can be found inside the "teiid-<version>-client.jar" file at the root called "teiid-client-settings.orig.properties". Extract this file, make a copy, change the property values required for the chosen SSL mode, and place this file in the client application’s classpath before the "teiid-<version>-client.jar" file.

SSL properties and definitions that can be set in a "teiid-client-settings.properties" file are shown below.

```
########################################
# SSL Settings
########################################

#
# The key store type.  Defaults to JKS
#

org.teiid.ssl.keyStoreType=JKS

#
```

```
# The key store algorithm, defaults to
# the system property "ssl.TrustManagerFactory.algorithm"
#

#org.teiid.ssl.algorithm=

#
# The classpath or filesystem location of the
# key store.
#
# This property is required only if performing 2-way
# authentication that requires a specific private
# key.
#

#org.teiid.ssl.keyStore=

#
# The key store password (not required)
#

#org.teiid.ssl.keyStorePassword=

#
# The key alias(not required, if given named certificate is used)
#

#org.teiid.ssl.keyAlias=

#
# The key password(not required, used if the key password is different than the keystore password)
#

#org.teiid.ssl.keyPassword=

#
# The classpath or filesystem location of the
# trust store.
#
# This property is required if performing 1-way
# authentication that requires trust not provided
# by the system defaults.
#

#org.teiid.ssl.trustStore=

#
# The trust store password (not required)
#

#org.teiid.ssl.trustStorePassword=

#
# The cipher protocol, defaults to TLSv3
#

org.teiid.ssl.protocol=TLSv1

#
# Whether to allow anonymous SSL
# (the TLS_DH_anon_WITH_AES_128_CBC_SHA cipher suite)
# defaults to true
#

org.teiid.ssl.allowAnon=true

#
# Whether to allow trust all server certificates
# defaults to false
```

```
#

#org.teiid.ssl.trustAll=false

#
# Whether to check for expired server certificates (no affect in anonymous mode or with trustAll=true)
# defaults to false
#

#org.teiid.ssl.checkExpired=false
```

1-way SSL

```
org.teiid.ssl.trustStore=<dir>/server.truststore (required)
```

2-way SSL

```
org.teiid.ssl.keyStore=<dir>/client.keystore (required)
org.teiid.ssl.trustStore=<dir>/server.truststore (required)
```

Additional Socket Client Settings

A `teiid-client-settings.properties` file can be used to configure Data Virtualization low level and SSL connection properties. Currently only a single properties file is expected per driver/classloader combination. A sample `teiid-client-settings.properties` file can be found inside the `teiid-<version>-client.jar` file at the root called `teiid-client-settings.orig.properties`. To customize the settings, extract this file, make a copy, change the property values accordingly, and place this file in the client application's classpath before the `teiid-<version>-client.jar" file`. Typically clients will not need to adjust the non-SSL properties. The following properties are available:

```
########################################
# Misc Socket Configuration
########################################

#
# The time in milliseconds for socket timeouts.
# Timeouts during the initialization, handshake, or
# a server ping may be treated as an error.
#
# This is the lower bound for all other timeouts
# the JDBC login timeout.
#
# Typically this should be left at the default of 1000
# (1 second). Setting this value too low may cause read
# errors.
#

org.teiid.sockets.soTimeout=1000

#
# Set the max time to live (in milliseconds) for non-execution
# synchronous calls.
#

org.teiid.sockets.synchronousttl=240000

#
# Set the socket receive buffer size (in bytes)
# 0 indicates that the default socket setting will be used.
#

org.teiid.sockets.receiveBufferSize=0

#
# Set the socket send buffer size (in bytes)
# 0 indicates that the default socket setting will be used.
#

org.teiid.sockets.sendBufferSize=0

#
# Set to true to enable Nagle's algorithm to conserve bandwidth
# by minimizing the number of segments that are sent.
#

org.teiid.sockets.conserveBandwidth=false

#
# Maximum number of bytes per server message.
# May need to be increased when using custom types and/or large batch sizes.
#

org.teiid.sockets.maxObjectSize=33554432
```

Note	All properties listed in "teiid-client-settings.properties" can also be set as System or env properties.

Prepared Statements

Teiid provides a standard implementation of `java.sql.PreparedStatement`. PreparedStatements can be very important in speeding up common statement execution, since they allow the server to skip parsing, resolving, and planning of the statement. See the Java documentation for more information on PreparedStatement usage.

`PreparedStatement` Considerations

- It is not necessary to pool client side Teiid `PreparedStatements`, since Teiid performs plan caching on the server side.
- The number of cached plans is configurable, and cached plans are purged by the least recently used (LRU). For information about configuring cached plans, see the Admin Guide.
- Cached plans are not distributed through a cluster. A new plan must be created for each cluster member.
- Plans are cached for the entire VDB or for just a particular session. The scope of a plan is detected automatically based upon the functions evaluated during it’s planning process.
- Stored procedures executed through a `CallableStatement` have their plans cached just as a `PreparedStatement`.
- Bind variable types in function signatures, e.g. "where t.col = abs(?)" can be determined if the function has only one signature or if the function is used in a predicate where the return type can be determined. In more complex situations it may be necessary to add a type hint with a cast or convert, e.g. upper(convert(?, string)).
- If you have the same value of a binding repeated multiple times in your query, you can consolidate that usage in a couple of ways.
  - The query can be enclosed as a anonymous procedure block:

```
BEGIN
  DECLARE string PARAM1 = cast(? as string);
  SELECT ... WHERE COLUMN1 = $1 AND COLUMN2 = $1 ...;
```

Note the cast of the bind variable, which is due to a small issue with the resolver that isn’t inferring the type from the variable declaration.

- You can also use the PostgreSQL like feature of $n positional bindings:

```
SELECT ... WHERE COLUMN1 = $1 AND COLUMN2 = $1 ...
```

ResultSet Limitations

The following limitations apply to result sets in Teiid:

- TYPE_SCROLL_SENSITIVE are not compatible.
- `UPDATABLE` ResultSets are not compatible.
- You cannot return multiple ResultSets from a Procedure execution.

JDBC Extensions

These are custom extensions to JDBC API from Teiid to provide compatibility with various features.

Statement Extensions

The Teiid statement extension interface, `org.teiid.jdbc.TeiidStatement` , provides functionality beyond the JDBC standard. To use the extension interface, simply cast or unwap the statement returned by the Connection. The following methods are provided on the extension interface:

Table 1. **Connection Properties**

Method Name	Description
`getAnnotations`	Get the query engine annotations if the statement was last executed with SHOWPLAN ON/DEBUG. Each `org.teiid.client.plan.Annotation` contains a description, a category, a severity, and possibly a resolution of notes recorded during query planning that can be used to understand choices made by the query planner.
`getDebugLog`	Get the debug log if the statement was last executed with SHOWPLAN DEBUG.
`getExecutionProperty`	Get the current value of an execution property on this statement object.
`getPlanDescription`	Get the query plan description if the statement was last executed with SHOWPLAN ON/DEBUG. The plan is a tree made up of `org.teiid.client.plan.PlanNode` objects. Typically `PlanNode.toString()` or `PlanNode.toXml()` will be used to convert the plan into a textual form.
`getRequestIdentifier`	Get an identifier for the last command executed on this statement. If no command has been executed yet, null is returned.
`setExecutionProperty`	Set the execution property on this statement. See the Execution Properties section for more information. It is generally preferable to use the SET Statement unless the execution property applies only to the statement being executed.
`setPayload`	Set a per-command payload to pass to translators. Currently the only built-in use is for sending hints for Oracle data source.

Partial Results Mode

You can use a "partial results" query mode with the Teiid Server. In this mode, the behavior of the query processor changes so that the server returns results even when some data sources are unavailable.

For example, suppose that two data sources exist for different suppliers and your data designers have created a virtual group that creates a union between the information from the two suppliers. If your application submits a query without using partial results query mode and one of the suppliers’ databases is down, the query against the virtual group returns an exception. However, if your application runs the same query in "partial results" query mode, the server returns data from the running data source and no data from the data source that is down.

When using "partial results" mode, if a source throws an exception during processing it does not cause the user’s query to fail. Rather, that source is treated as returning no more rows after the failure point. Most commonly, that source will return 0 rows.

This behavior is most useful when using `UNION` or `OUTER JOIN` queries as these operations handle missing information in a useful way. Most other kinds of queries will simply return 0 rows to the user when used in partial results mode and the source is unavailable.

For each source that is excluded from the query, a warning will be generated describing the source and the failure. These warnings can be obtained from the `Statement.getWarnings()` method. This method returns a `SQLWarning` object but in the case of "partial results" warnings, this will be an object of type `org.teiid.jdbc.PartialResultsWarning` class. This class can be used to obtain a list of all the failed sources by name and to obtain the specific exception thrown by each source.

Note	Because Teiid enables cursoring before an entire result is formed, it is possible that a data source failure will not be determined until after the first batch of results have been returned to the client. This can happen in the case of unions, but not joins. To ensure that all warnings have been accumulated, the statement should be checked after the entire result set has been read.

Note	If other warnings are returned by execution, then the partial results warnings may occur after the first warning in the warning chain.

Partial results mode is off by default but can be turned on for all queries in a Connection with either setPartialResultsMode("true") on a DataSource or partialResultsMode=true on a JDBC URL. In either case, partial results mode may be toggled later with a SET Statement.

Setting Partial Results Mode

```
Statement statement = ...obtain statement from Connection...
statement.execute("set partialResultsMode true");
```

Getting Partial Results Warnings

```
statement.execute("set partialResultsMode true");
ResultSet results = statement.executeQuery("SELECT Name FROM Accounts");
while (results.next()) {
  ... //process the result set
}
SQLWarning warning = statement.getWarnings();
while(warning != null) {
  if (warning instanceof PartialResultsWarning) {
    PartialResultsWarning partialWarning = (PartialResultsWarning)warning;
    Collection failedConnectors = partialWarning.getFailedConnectors();
    Iterator iter = failedConnectors.iterator();
    while(iter.hasNext()) {
      String connectorName = (String) iter.next();
      SQLException connectorException = partialWarning.getConnectorException(connectorName);
      System.out.println(connectorName + ": " + connectorException.getMessage());
    }
```

```
    }
    warning = warning.getNextWarning();
  }
```

Warning	In some instances, typically JDBC sources, the source not being initially available will prevent Teiid from automatically determining the appropriate set of source capabilities. If you get an exception indicating that the capabilities for an unavailable source are not valid in partial results mode, then it may be necessary to manually set the database version or similar property on the translator to ensure that the capabilities are known even if the source is not available.

Non-blocking Statement Execution

JDBC query execution can indefinitely block the calling thread when a statement is executed or a resultset is being iterated. In some situations you may not wish to have your calling threads held in these blocked states. When using embedded/local connections, you may optionally use the `org.teiid.jdbc.TeiidStatement` and `org.teiid.jdbc.TeiidPreparedStatement` interfaces to execute queries with a callback `org.teiid.jdbc.StatementCallback` that will be notified of statement events, such as an available row, an exception, or completion. Your calling thread will be free to perform other work. The callback will be executed by an engine processing thread as needed. If your results processing is itself blocking and you want query processing to be concurrent with results processing, then your callback should implement onRow handling in a multi-threaded manner to allow the engine thread to continue.

Non-blocking Prepared Statement Execution

```
PreparedStatement stmt = c.prepareStatemen(sql);
TeiidPreparedStatement tStmt = stmt.unwrap(TeiidPreparedStatement.class);
tStmt.submitExecute(new StatementCallback() {
    @Override
    public void onRow(Statement s, ResultSet rs) {
        //any logic that accesses the current row ...
        System.out.println(rs.getString(1));
    }

    @Override
    public void onException(Statement s, Exception e) throws Exception {
        s.close();
    }

    @Override
    public void onComplete(Statement s) throws Exception {
        s.close();
    }, new RequestOptions()
});
```

The non-blocking logic is limited to statement execution only. Other JDBC operations, such as connection creation or batched executions do not yet have non-blocking options.

If you access forward positions in the onRow method (calling next, isLast, isAfterLast, absolute), they may not yet be valid and a `org.teiid.jdbc.AsynchPositioningException` will be thrown. That exception is recoverable if caught or can be avoided by calling `TeiidResultSet.available()` to determine if your desired positioning will be valid.

Continuous Execution

The `RequestOptions` object may be used to specify a special type of continuous asynch execution via the `continuous` or `setContinuous` methods. In continuous mode the statement will be continuously re-executed. This is intended for consuming real-time or other data streams processed through a SQL plan. A continuous query will only terminate on an error or when the statement is explicitly closed. The SQL for a continuous query is no different than any other statement. Care should be taken to ensure that retrievals from non-continuous sources is appropriately cached for reuse, such as by using materialized views or session scoped temp tables.

A continuous query must do the following:

- return a result set
- be executed with a forward-only result set
- cannot be used in the scope of a transaction

Since resource consumption is expected to be different in a continuous plan, it does not count against the server max active plan limit. Typically custom sources will be used to provide data streams.

For more information, see *ReusableExecutions* in the Developers Guide.

When the client wishes to end the continuous query, the `Statement.close()` or `Statement.cancel()` method should be called. Typically your callback will close whenever it no long needs to process results.

See also the `ContinuousStatementCallback` for use as the `StatementCallback` for additional methods related to continuous processing.

ResultSet Extensions

The Teiid result set extension interface, `org.teiid.jdbc.TeiidResultSet`, provides functionality beyond the JDBC standard. To use the extension interface, simply cast or unwrap a result set returned by a Teiid statement. The following methods are provided on the extension interface:

Table 1. **Connection Properties**

Method Name	Description
`available`	Returns an estimate of the minimum number of rows that can be read (after the current) without blocking or the end of the ResultSet is reached.

Connection Extensions

Teiid connections (defined by the `org.teiid.jdbc.TeiidConnection` interface) are compatible with the changeUser method to reauthenticate a given connection. If the reauthentication is successful the current connection my be used with the given identity. Existing statements/result sets are still available for use under the old identity.

See the JBossAS issue JBAS-1429 for more on using reauthentication support with JCA.

Incompatible JDBC Methods

Based upon the JDBC in JDK 1.6, this appendix details only those JDBC methods that Teiid is not compatible with. Unless specified below, Teiid is compatible with all other JDBC Methods.

Those methods listed without comments throw a SQLException stating that it is not supported.

Where specified, some listed methods do not throw an exception, but possibly exhibit unexpected behavior. If no arguments are specified, then all related (overridden) methods are not compatible. If an argument is listed then only those forms of the method specified are not compatible.

Incompatible Classes and Methods in "java.sql"

Class name	Methods
Blob	getBinaryStream(long, long) - throws SQLFeatureNotSupportedException setBinaryStream(long) - - throws SQLFeatureNotSupportedException setBytes - - throws SQLFeatureNotSupportedException truncate(long) - throws SQLFeatureNotSupportedException
CallableStatement	getObject(int parameterIndex, Map<String, Class<?>> map) - throws SQLFeatureNotSupportedException getRef - throws SQLFeatureNotSupportedException getRowId - throws SQLFeatureNotSupportedException getURL(String parameterName) - throws SQLFeatureNotSupportedException registerOutParameter - ignores registerOutParameter(String parameterName, *) - throws SQLFeatureNotSupportedException setRowId(String parameterName, RowId x) - throws SQLFeatureNotSupportedException setURL(String parameterName, URL val) - throws SQLFeatureNotSupportedException
Clob	getCharacterStream(long arg0, long arg1) - throws SQLFeatureNotSupportedException setAsciiStream(long arg0) - throws SQLFeatureNotSupportedException setCharacterStream(long arg0) - throws SQLFeatureNotSupportedException setString - throws SQLFeatureNotSupportedException truncate - throws SQLFeatureNotSupportedException
Connection	createBlob - throws SQLFeatureNotSupportedException createClob - throws SQLFeatureNotSupportedException createNClob - throws SQLFeatureNotSupportedException createSQLXML - throws SQLFeatureNotSupportedException createStruct(String typeName, Object[] attributes) - throws SQLFeatureNotSupportedException getClientInfo - throws SQLFeatureNotSupportedException releaseSavepoint - throws SQLFeatureNotSupportedException rollback(Savepoint savepoint) - throws SQLFeatureNotSupportedException setHoldability - throws SQLFeatureNotSupportedException setSavepoint - throws SQLFeatureNotSupportedException setTypeMap - throws SQLFeatureNotSupportedException setRealOnly - effectively ignored
DatabaseMetaData	getAttributes - throws SQLFeatureNotSupportedException getClientInfoProperties - throws SQLFeatureNotSupportedException getRowIdLifetime - throws SQLFeatureNotSupportedException
NClob	Not Supported
PreparedStatement	setRef - throws SQLFeatureNotSupportedException setRowId - throws SQLFeatureNotSupportedException setUnicodeStream - throws SQLFeatureNotSupportedException
Ref	Not Implemented
	deleteRow - throws SQLFeatureNotSupportedException getHoldability - throws SQLFeatureNotSupportedException getObject(*, Map<String, Class<?>> map) - throws SQLFeatureNotSupportedException

ResultSet	getRef - throws SQLFeatureNotSupportedException getRowId - throws SQLFeatureNotSupportedException getUnicodeStream - throws SQLFeatureNotSupportedException getURL - throws SQLFeatureNotSupportedException insertRow - throws SQLFeatureNotSupportedException moveToInsertRow - throws SQLFeatureNotSupportedException refreshRow - throws SQLFeatureNotSupportedException rowDeleted - throws SQLFeatureNotSupportedException rowInserted - throws SQLFeatureNotSupportedException rowUpdated - throws SQLFeatureNotSupportedException setFetchDirection - throws SQLFeatureNotSupportedException update* - throws SQLFeatureNotSupportedException
RowId	Not Supported
Savepoint	not Supported
SQLData	Not Supported
SQLInput	not Supported
SQLOutput	Not Supported

Incompatible Classes and Methods in "javax.sql"

Class name	Methods
`RowSet*`	`Not Supported`

ODBC compatibility

Open Database Connectivity (ODBC) is a standard database access method developed by the SQL Access group in 1992. ODBC, just like JDBC in Java, allows consistent client access regardless of which database management system (DBMS) is handling the data. ODBC uses a driver to translate the application's data queries into commands that the DBMS understands. For this to work, both the application and the DBMS must be ODBC-compliant – that is, the application must be capable of issuing ODBC commands and the DBMS must be capable of responding to them.

Teiid can provide ODBC access to deployed VDBs in the Teiid runtime through PostgreSQL's ODBC driver. This is possible because Teiid has a PostgreSQL server emulation layer accessible via socket clients.

Note	By default, ODBC is enabled and running on on port *35432*.

The pg emulation is not complete. The intention of the ODBC access is to provide non-JDBC connectivity to issue Teiid queries - not pgsql queries. Although you can use many PostgreSQL constructs, the default behavior for queries matches Teiid's expectations. See System Properties for optional properties that further emulate pgsql handling.

Note	**Handling names with underscore ("*")* in ODBC. By default Teiid does not have a default like escape character. Depending upon the ODBC client however there may be an expectation that backslash is used by default - which is the behavior of PostgreSQL. This may cause metadata queries to be issued against objects with "" in their name to return no or incorrect results. You may globally emulate the behavior of PostgreSQL by setting the `org.teiid.backslashDefaultMatchEscape` system property to `true` . To alter the property just for the current session then have your ODBC client issue `select cast(teiid_session_set('backslashDefaultMatchEscape', true) as boolean)` statement before any other statement.

Postgres ODBC drivers 9.5 and later do not require this special property as the client will use an E escaped literal instead.

Compatibility was last ensured with the 9.6 Postgres ODBC driver. You are encouraged to use later client versions when needed and report any issues to the community.

Known Limitations:

- Updateable cursors are not supported. You will receive parsing errors containing the pg system column ctid if this feature is not disabled.
- LO support is not available. LOBs will be returned as string or bytea as appropriate using the transport max lob size setting.
- The Teiid object type will map to the PostgreSQL UNKNOWN type, which cannot be serialized by the ODBC layer. Cast/Convert should be used to provide a type hint when appropriate - for example teiid_session_set returns an object value. "SELECT teiid_session_set('x', 'y')" will fail, but "SELECT cast(teiid_session_set('x', 'y') as string)" will succeed.
- Multi-dimensional arrays are not supported.

Installation

Before an application can use ODBC, you must first install the ODBC driver on same machine that the application is running on and then create a Data Source Name (DSN) that represents a connection profile for your Teiid VDB.

For a Windows client, see the Windows Installation Guide.

Configuration

Warning	By default, clients use plain text password authentication in Teiid for pg/ODBC interfaces. If the client/server are not configured to use SSL or GSS authentication, the password will be sent in plain text over the network.

For a Windows client, see Configuring the Data Source Name.

See also DSN Less Connection.

Connection Settings

All the available pg driver connection options with their descriptions that can be used are defined here https://odbc.postgresql.org/docs/config.html. When using these properties on the connection string, their property names are defined here https://odbc.postgresql.org/docs/config-opt.html.

However Teiid does not honor all properties, and some, such as Updatable Cursors, will cause query failures.

Table 1. **Primary ODBC Settings For Teiid**

Name	Description
Updateable Cursors & Row Versioning	Should not be used.
Use serverside prepare & Parse Statements & Disallow Premature	It is recommended that "Use serverside prepare" is enabled and "Parse Statements"/"Disallow Premature" are disabled
SSL mode	May be needed if you are connecting to a secured pg transport port. See Security Guide
Use Declare/Fetch cursors & Fetch Max Count	Should be used to better manage resources when large result sets are used

Logging/debug settings can be utilized as needed.

Settings that manipulate datatypes, metadata, or optimizations such as "Show SystemTables", "True is -1", "Backend genetic optimizer", "Bytea as LongVarBinary", "Bools as Char", etc. are ignored by the Teiid server and have no client side effect. If there is a need for these or any other settings to have a defined affect, please open an issue with the product/project.

Any other setting that does have a client side affect, such as "LF <→ CR/LF conversion", may be used if desired but there is currently no server side usage of the setting.

Teiid Connection Settings

Most Teiid specific connection properties do not map to ODBC client connection settings. If you find yourself in this situation and cannot use post connection SET statements, then you can set default ODBC connection properties for the virtual database. Use VDB properties of the form `connection.XXX` to control things like partial results mode, result set caching, etc.

The application name may be set by some clients. If not, you may use a SET statement - "SET application_name name" - to set the name even after the connection is made.

Installing the ODBC Driver Client

A PostgreSQL ODBC driver needed to make the ODBC connection to Teiid is *not* bundled with the Teiid distribution. The appropriate driver needs be downloaded directly from the PostgreSQL web site. The *8.04.200* version of the ODBC driver was extensively tested for compatibility.

Microsoft Windows

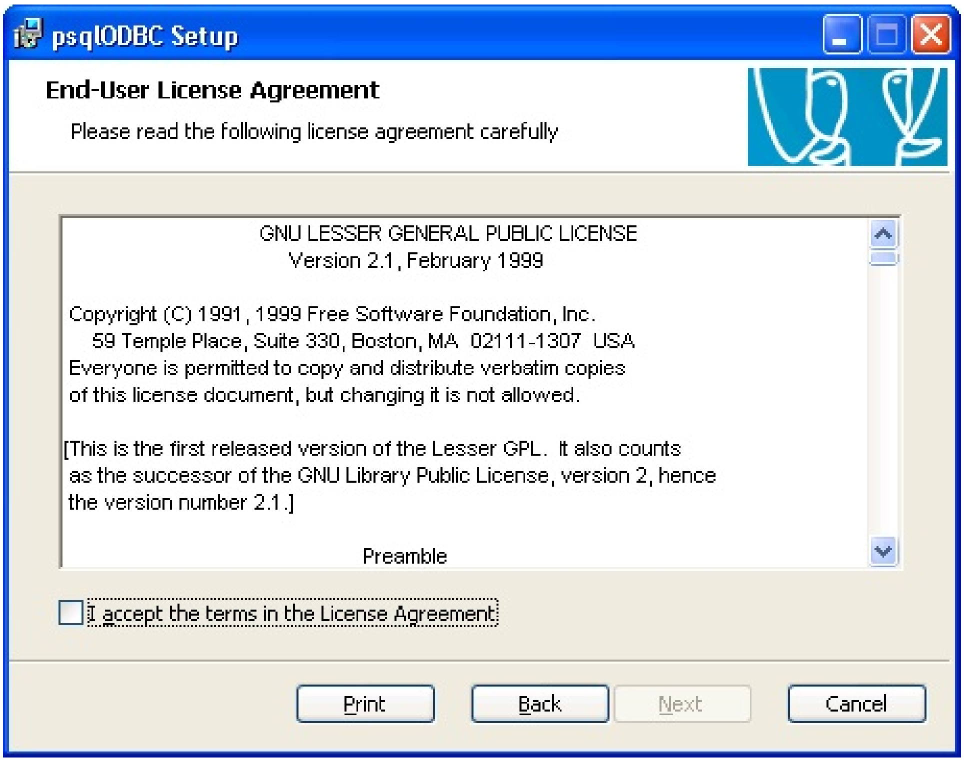
1. Download at least the ODBC 8.4 driver from the PostgreSQL download site. If you are looking for 64-bit Windows driver download the driver from here. Later versions of the driver may be used the 9.0-9.5 clients have been used extensively by the Teiid community. There are no active issues against 9.6 and later, but they have not yet seen as much use - if you encounter an issue, please create a JIRA.

2. Extract the contents of the ZIP file into a temporary location on your system. For example: "c:\temp\pgodbc"

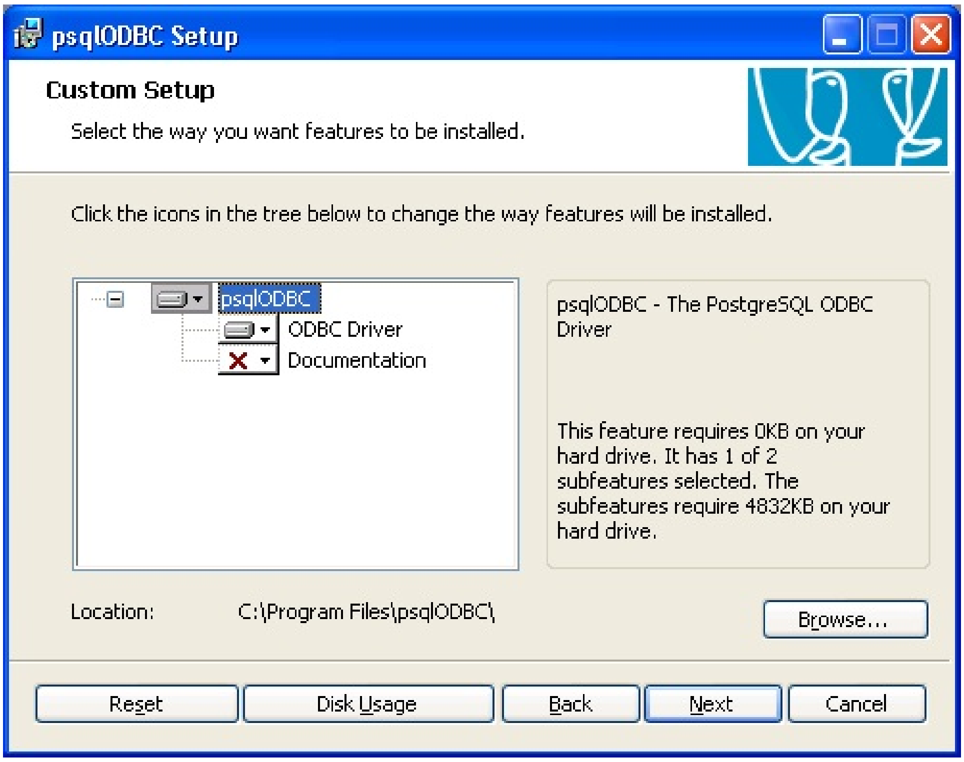
3. Double click on "psqlodbc.msi" file or (.exe file in the case of 64 bit) to start installation of the driver.

4. The Wizard appears as

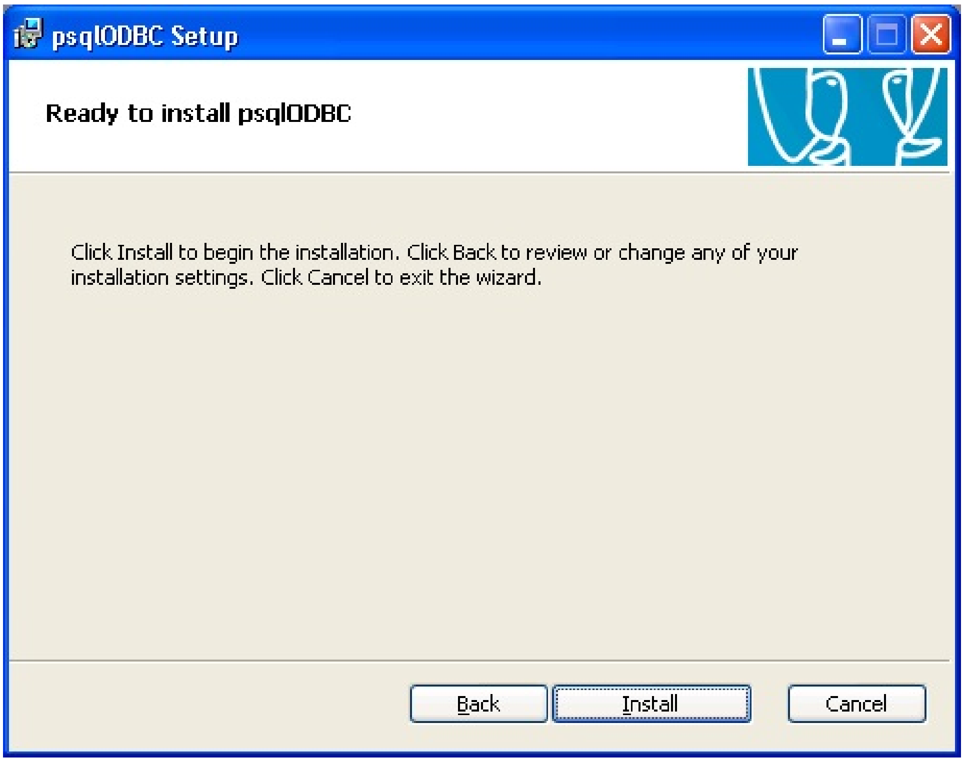
Click "Next". 5. The next step of the wizard displays.



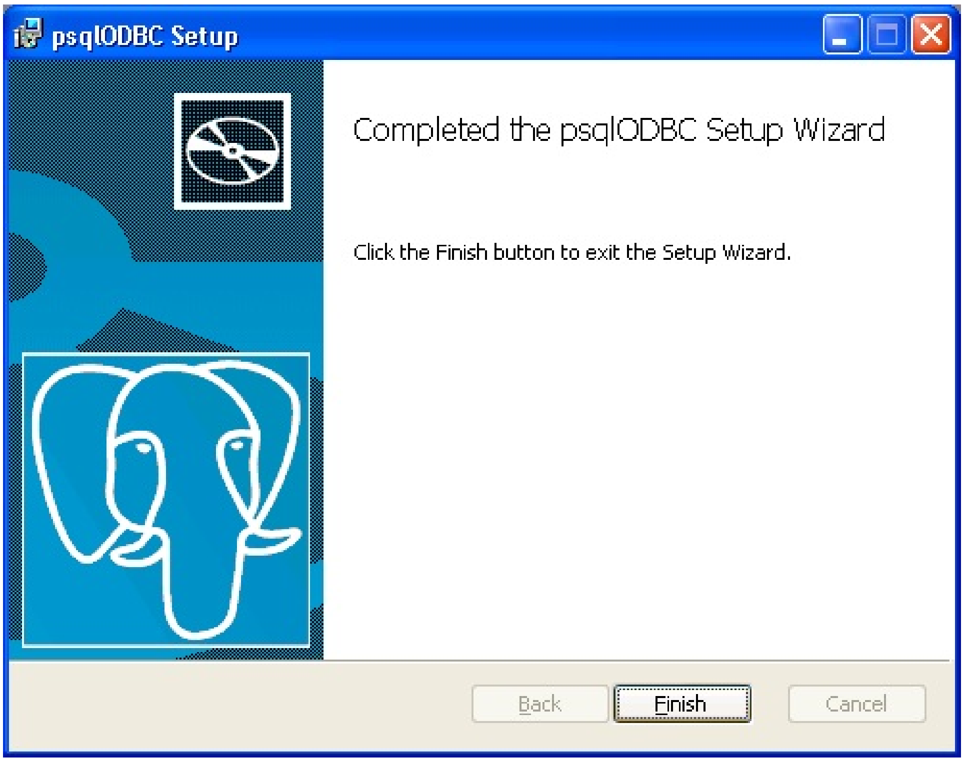
Carefully read it, and check the "I accept the terms in the License Agreement", if you are agreeing to the licensing terms. Then click "Next". 6. The next step of the wizard displays.



If you want to install in a different directory than the default that is already selected, click the "Browse" button and select a directory. Click "Next" to start installing in the selected directory. 7. The next step of the wizard displays.



This step summarizes the choices you have made in the wizard. Review this information. If you need to change anything, you can use the Back button to return to previous steps. Click "Install" to proceed. 8. 1.The installation wizard copies the necessary files to the location you specified. When it finishes, the following screen displays.



Click "Finish" to complete.

Other *nix Platform Installations

For all other platforms other than Microsoft Windows, the ODBC driver needs built from the source files provided. Download the ODBC driver source files from the PostgreSQL download site. Untar the files to a temporary location. For example: "~/tmp/pgodbc". Build and install the driver by running the commands below.

Note	You should use super user account or use "sudo" command for running the "make install" command.

```
% tar -zxvf psqlodbc-xx.xx.xxxx.tar.gz
% cd psqlodbc-xx.xx.xxxx
% ./configure
% make
% make install
```

Some *nix distributions may already provide binary forms of the appropriate driver, which can be used as an alternative to building from source.

Configuring the Data Source Name (DSN)

See Teiid compatible options for a description of the available client configuration options.

Windows Installation

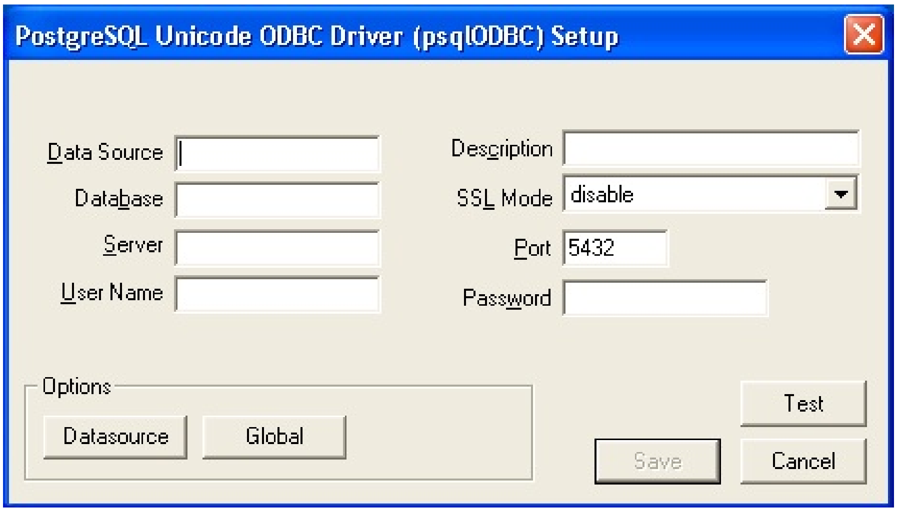
Once you have installed the ODBC Driver Client software on your workstation, you have to configure it to connect to a Teiid Runtime. Note that the following instructions are specific to the Microsoft Windows Platform.

To do this, you must have logged into the workstation with administrative rights, and you need to use the Control Panel's *Data Sources (ODBC)* applet to add a new data source name.

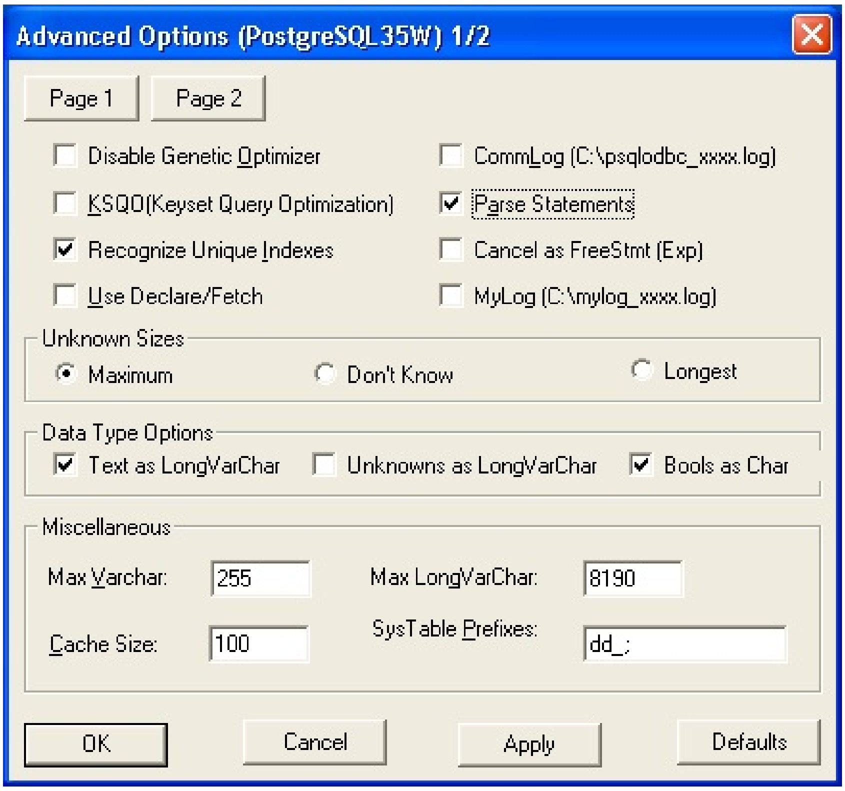
Each data source name you configure can only access one VDB within a Teiid System. To make more than one VDB available, you need to configure more than one data source name.

Follow the below steps in creating a data source name (DSN)

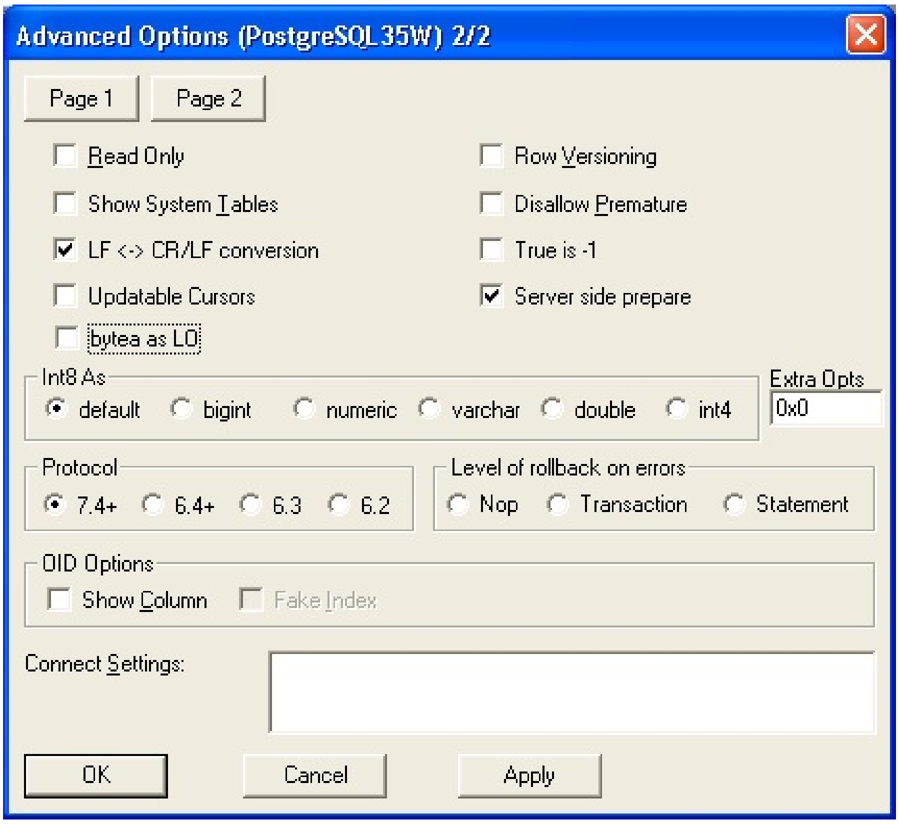
1. From the Start menu, select Settings > Control Panel.
2. The Control Panel displays. Double click *Administrative Tools*.
3. Then Double-click *Data Sources (ODBC)*.
4. The ODBC Data Source Administrator applet displays. Click the tab associated with the type of DSN you want to add.
5. The Create New Data Source dialog box displays. In the Select a driver for which you want to set up a data source table, select *PostgreSQL Unicode*.
6. Click Finish
7. The PostgreSQL ODBC DSN Setup dialog box displays.

   In the *Data Source* Name edit box, type the name you want to assign to this data source. In the *Database* edit box, type the name of the virtual database you want to access through this data source. In the *Server* edit box, type the host name or IP address of your Teiid runtime. If connecting via a firewall or NAT address, the firewall address or NAT address should be entered. In the Port edit box, type the port number to which the Teiid System listens for ODBC requests. By default, Teiid listens for ODBC requests on port 35432 In the *User Name* and *Password* edit boxes, supply the user name and password for the Teiid runtime access. Provide any description about the data source in the *Description* field.
8. Click on the *Datasource* button, you will see this below figure. Configure options as shown.



Click on "page2" and make sure the options are selected as shown



9. Click "save" and you can optionally click "test" to validate your connection if the Teiid is running. You have configured a Teiid's virtual database as a data source for your ODBC applications. Now you can use applications such as Excel, Access to query the data in the VDB

Other *nix Platform Installations

Before you can access Teiid using ODBC on any *nix platforms, you need to either install a ODBC driver manager or verify that one already exists. As the ODBC Driver manager Teiid recommends unixODBC. If you are working with RedHat Linux or Fedora you can check the graphical "yum" installer to search, find and install unixODBC. Otherwise you can download the unixODBC manager here. To install, simply untar the contents of the file to a temporary location and execute the following commands as super user.

```
./configure
make
make install
```

Check unixODBC website site for more information, if you run into any issues during the installation.

Now, to verify that PostgreSQL driver installed correctly from earlier step, execute the following command

```
odbcinst -q -d
```

That should show you all the ODBC drivers installed in your system. Now it is time to create a DSN. Edit "/etc/odbc.ini" file and add the following

```
  [<DSN name>]
  Driver = /usr/lib/psqlodbc.so
  Description = PostgreSQL Data Source
  Servername = <Teiid Host name or ip>
  Port = 35432
  Protocol = 7.4-1
  UserName = <user-name>
  Password = <password>
  Database = <vdb-name>
  ReadOnly = no
  ServerType = Postgres
  ConnSettings =
  UseServerSidePrepare=1
  Debug=0
  Fetch = 10000
  # enable below when dealing large resultsets to enable cursoring
  #UseDeclareFetch=1
```

Note that you need "sudo" permissions to edit the "/etc/odbc.ini" file. For all the available configurable options that you can use in defining a DSN can be found here on postgreSQL ODBC page.

Once you are done with defining the DSN, you can verify your DSN using the following command

```
isql <DSN-name> [<user-name> <password>] < commands.sql
```

where "commands.sql" file contains the SQL commands you would like to execute. You can also omit the commands.sql file, then you will be provided with a interactive shell.

Tip	You can also use languages like Perl, Python, C/C++ with ODBC ports to Postgres, or if they have direct Postgres connection modules you can use them too to connect Teiid and issue queries an retrieve results.

DSN Less Connection

You can also connect to Teiid VDB using ODBC with out explicitly creating a DSN. However, in these scenarios your application needs, what is called as "DSN less connection string". The below is a sample connection string

For Windows:

```
ODBC;DRIVER={PostgreSQL Unicode};DATABASE=<vdb-name>;SERVER=<host-name>;PORT=<port>;Uid=<username>;Pwd=<password>;c4=0;c8=1;
```

For *nix:

```
ODBC;DRIVER={PostgreSQL};DATABASE=<vdb-name>;SERVER=<host-name>;PORT=<port>;Uid=<username>;Pwd=<password>;c4=0;c8=1;
```

See the available Teiid ODBC connection settings.

Configuring Connection Properties with ODBC

When working with ODBC connections, you can set the URL connection properties that are available in Teiid by running a command such as the following:

```
SET <property-name> TO <property-value>
```

For example, to turn on result set caching you can run the following command:

```
SET resultSetCacheMode TO 'true'
```

Another option is to use VDB properties in the vdb file to configure the connection, as in the following example:

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

Or in an XML VDB:

```
<vdb name="...">
    <property name="connection.resultSetCacheMode" value="true"/>
    ...
</vdb>
```

OData compatibility

What is OData

The Open Data Protocol (OData) is a Web protocol 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. The protocol emerged from experiences implementing AtomPub clients and servers in a variety of products over the past several years. OData is 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.

OData is consistent with the way the Web works - it makes a deep commitment to URIs for resource identification and commits to an HTTP-based, uniform interface for interacting with those resources (just like the Web). This commitment to core Web principles allows OData to enable a new level of data integration and interoperability across a broad range of clients, servers, services, and tools.

copied from http://odata.org

Teiid compatibility for OData

Teiid is compatible with OData Version 4.0.

When a user successfully deploys a VDB into a Teiid Server, the OData protocol compatibility is implicitly provided by the Teiid server without any further configuration.

OData support is currently not available in the Teiid Embedded profile.

OData support is implemented and deployed through WAR file(s). Access is similar to accessing to any web resources deployed on the container. The war file(s) are located at <container root>/modules/org/jboss/teiid/deployments/*.war.

Legacy OData Version 2.0 support has been removed, but could be maintained as it’s own project - please contact the community if you still need this feature and want to maintain it.

OData Version 4.0 compatibility

Teiid strives to be compliant with the OData specification. The rest of this chapter highlight some specifics of OData and Teiid's compatibility, but you should also consult the specification.

Table of Contents

How to Access the data?

For example, if you have a vdb by name *northwind* deployed that has a *customers* table in a *NW* model, then you can access that table with an HTTP GET via the URL:

All users are granted an odata role by default, which allows all authenticated users to access odata end points. In previous versions of Teiid this role had to be manually assigned. If you still want that behavior remove the Identity login module from the teiid-security security domain that is granting the odata role.

```
http://localhost:8080/odata4/northwind/NW/customers
```

this would be akin to making a JDBC/ODBC connection and issuing the SQL:

```sql
SELECT * FROM NW.customers
```

Note	Use correct case (upper or lower) in the resource path. Unlike SQL, the names used in the URI as case-sensitive.

The returned results from OData query can be in Atom/AtomPub XML or JSON format. JSON results are returned by default.

Query Basics

Users can submit predicates with along their query to filter the results:

```
http://localhost:8080/odata4/northwind/NW/customers?$filter=name eq 'bob'
```

Note	Spaces around 'eq' are for readability of the example only; in real URLs they must be percent-encoded as %20. OData mandates percent encoding for all spaces in URLs. http://docs.oasis-open.org/odata/odata/v4.0/odata-v4.0-part2-url-conventions.html

this would be similar to making a JDBC/ODBC connection and issuing the SQL

```
SELECT * FROM NW.customers where name = 'bob'
```

To request the result to be formatted in a specific format, add the query option $format

```
http://localhost:8080/odata4/northwind/NW/customers?$format=JSON
```

Query options can be combined as needed. For example format with a filter:

```
http://localhost:8080/odata4/northwind/NW/customers?$filter=name eq 'bob'&$format=xml
```

OData allows for querying navigations from one entity to another. A navigation is similar to the foreign key relationships in relational databases.

For example, if the *customers* table has an exported key to the *orders* table on the *customers* primary key called the customer_fk, then an OData GET could be issued like:

```
http://localhost:8080/odata4/northwind/NW/customers(1234)/customer_fk?$filter=orderdate gt datetime'2012-12-31T
21:23:38Z'
```

this would be akin to making a JDBC/ODBC connection and issuing the SQL:

```
SELECT o.* FROM NW.orders o join NW.customers c on o.customer_id = c.id where c.id=1234 and o.orderdate > {ts '
2012-12-31 21:23:38'}
```

Note	**More Comprehensive Documentation about ODATA** - For detailed protocol access you can read the specification at http://odata.org. You can also read this very useful web resource for an example of accessing an OData server.

How to execute a stored procedure?

Odata allows you to call your exposed stored procedure methods via odata.

```
http://localhost:8080/odata4/northwind/NW/getcustomersearch(id=120,firstname='miche
al')
```

Not seeing all the rows?

See the configuration section below for more details. Generally batching is being utilized, which tooling should understand automatically, and additional queries with a $skiptoken query option specified are needed:

```
http://localhost:8080/odata4/northwind/NW/customers?$skiptoken=xxx
```

"EntitySet Not Found" error?

When you issue the above query are you seeing a message similar to below?

```
{"error":{"code":null,"message":"Cannot find EntitySet, Singleton, ActionImport or FunctionImport with name 'xx
x'."}}
```

Then, it means that either you supplied the model-name/table-name combination wrong, check the spelling and case.

It is possible that the entity is not part of the metadata, such as when a table does not have any PRIMARY KEY or UNIQUE KEY(s).

How to update your data?

Using the OData protocol it is possible to perform CREATE/UPDATE/DELETE operations along with READ operations shown above. These operations use different HTTP methods.

INSERT/CREATE is accomplished through an HTTP method "POST".

Example POST

```
POST /service.svc/Customers HTTP/1.1
Host: host
Content-Type: application/json
Accept: application/json
{
  "CustomerID": "AS123X",
  "CompanyName": "Contoso Widgets",
  "Address" : {
     "Street": "58 Contoso St",
     "City": "Seattle"
  }
}
```

An UPDATE is performed with an HTTP "PUT".

Example PUT Update of Customer

```
PUT /service.svc/Customers('ALFKI') HTTP/1.1
Host: host
Content-Type: application/josn
Accept: application/json
{
  "CustomerID": "AS123X",
  "CompanyName": "Updated Company Name",
  "Address" : {
     "Street": "Updated Street"
  }
}
```

The DELETE operation uses the HTTP "DELETE" method.

Example Delete

```
DELETE /service.svc/Customers('ALFKI') HTTP/1.1
Host: host
Content-Type: application/json
Accept: application/json
```

Security

By default OData access is secured using HTTPBasic authentication. The user will be authenticated against Teiid’s default security domain "teiid-security".

However, if you wish to change the security domain use a deployment-overlay to override the *web.xml* file in the odata4 file in the *<modules>/org/jboss/teiid/main/deployments* directory.

OData WAR is also compatible with Kerberos, SAML and OAuth2 authentications. For information about configuring these security schemes, see Security Guide

Configuration

The OData WAR file can be configured with following properties in the web.xml file.

Property Name	Description	Default Value
batch-size	Number of rows to send back each time, -1 returns all rows	256
skiptoken-cache-time	Time interval between the results being recycled/expired between $skiptoken requests	300000
invalid-xml10-character-replacement	XML 1.0 replacement character for non UTF-8 characters.	
local-transport-name	Teiid Local transport name for connection	odata
invalid-xml10-character-replacement	Replacement string if an invalid XML 1.0 character appears in the data - note that this replacement will occur even if JSON is requested. No value (the default) means that an exception will be thrown with XML results if such a character is encountered.	
proxy-base-uri	Defines the proxy server’s URI to be used in OData responses. Only needs to be set for proxies that do not support the Forwarded nor the X-Forwarded headers.	n/a
connection.XXX	Sets XXX as an execution property on the local connection. Can be used for example to enable result set cache mode.	n/a
explicit-vdb-version	When explicit-vdb-version is true, an explicit vdb version needs to be part of the url to use anything other than the default version 1 vdb. When explicit-vdb-version is false, the odata vdb version will be determined just like a JDBC connection.	true

Note	**"Behind Proxy or In Cloud Environments?"** - If the Teiid server is configured behind a proxy server or deployed in cloud environment, or using a load-balancer that does not support the Forwarded nor X-Forwarded headers, then the URI of the server which is handling the OData request is different from URI of proxy. To generate valid links in the OData responses configure "proxy-base-uri" property in the web.xml. If this value is available as system property then define the property value like below

```
<init-param>
    <param-name>proxy-base-uri</param-name>
    <param-value>${system-property-name}</param-value>
```

```
    </init-param>
```

To modify the web.xml, create a deployment-overlay using the cli with the modified contents:

```
deployment-overlay add --name=myOverlay --content=/WEB-INF/web.xml=/modified/web.xml --deployments=teiid-odata-odata4.war --redeploy-affected
```

Teiid OData server implements cursoring logic when the result rows exceed the configured batch size. On every request, only *batch-size* number of rows are returned. Each such request is considered an active cursor, with a specified amount of idle time specified by *skip-token-cache-time*. After the cursor is timed out, the cursor will be closed and remaining results will be cleaned up, and will no longer be available for further queries. Since there is no session based tracking of these cursors, if the request for skiptoken comes after the expired time, the original query will be executed again and tries to reposition the cursor to relative absolute position, however the results are not guaranteed to be same as the underlying sources may have been updated with new information meanwhile.

Limitations

The OData4 interface is subject to some feature limitations. You cannot use the following features.

- Search.
- Delta processing.
- Data-aggregation extension of the OData specification.
- $it usage is limited to only primitive collection properties.

Client Tools for Access

OData access is really where the user comes in, depending upon your programming model and needs there are various ways you write your access layer into OData. The following are some suggestions:

- Your Browser: The OData Explorer is an online tool for browsing an OData data service.
- Olingo: Is a Java framework that supports OData V4, has both consumer and producer framework.
- Microsoft has various .Net based libraries, see http://odata.github.io/
- Windows Desktop: LINQPad is a wonderful tool for building OData queries interactively. See https://www.linqpad.net/
- Shell Scripts: use CURL tool

For latest information other frameworks and tools available please see http://www.odata.org/ecosystem/

OData Metadata (How Teiid interprets the relational schema into OData’s $metadata)

OData defines its schema using Conceptual Schema Definition Language (CSDL). A VDB in an ACTIVE state in Teiid exposes its visible metadata in CSDL format. For example if you want retrieve metadata for your vdb, you need to issue a request like:

```
http://localhost:8080/odata4/northwind/NW/$metadata
```

Since OData schema model is not a relational schema model, Teiid uses the following semantics to map its relational schema model to OData schema model.

Relational Entity	Mapped OData Entity
Model Name	Schema Namespace, EntityContainer Name
Table/View	EntityType, EntitySet
Table Columns	EntityType’s Properties
Primary Key	EntityType’s Key Properties
Foreign Key	Navigation Property on EntityType
Procedure	FunctionImport, ActionImport
Procedure’s Table Return	ComplexType

Teiid by design does not define any "embedded" ComplexType in the EntityType.

Since OData access is more key based, it is **MANDATORY** that every table Teiid exposes through OData must have a PK or at least one UNIQUE key. A table which does not either of these will be dropped out of the $metadata.

Since all data roles are not consulted in the construction of the OData metadata there are times when tables or procedures will need to be specifically hidden. This can be done in the vdb via a "teiid_odata:visible" extension metadata property on the object.

```
create foreign table HIDDEN (id long primary key, ...) OPTIONS ("teiid_odata:visible" false);
```

With teiid_odata:visible set to false the OData layer will not expose the given object.

Datatype Mapping

Teiid Type	OData Type
STRING	Edm.String
BOOLEAN	Edm.Boolean
BYTE	Edm.SByte
SHORT	Edm.Int16
INTEGER	Edm.Int32
LONG	Edm.Int64
FLOAT	Edm.Single
DOUBLE	Edm.Double
BIG_INTEGER	Edm.Decimal
BIG_DECIMAL	Edm.Decimal

DATE	Edm.Date
TIME	Edm.TimeOfDay
TIMESTAMP	Edm.DateTimeOffset
BLOB	Edm.Stream
CLOB	Edm.Stream
XML	Edm.Stream
VARBINARY	Edm.Binary

Geography and Geometry will be mapped to the corresponding Edm.GeometryXXX and Edm.GeographyXXX types based upon the associated {http://www.teiid.org/translator/spatial/2015}type property. A general mapping to Edm.Geometry or EdmGeography will fail to serialize the values correctly.

Where possible, array types will be mapped to a collection type. However you cannot include multidimensional arrays. Also array/collection values cannot be used as parameters nor in comparisons.

Functions And Actions

The mapping of entities and their properties is relatively straight-forward. The mapping of Teiid procedures to OData Functions and Actions is more involved. Virtual procedures, source procedure, and virtual functions defined by DDL (not a Java class) are all eligible to be mapped. Source functions or virtual functions defined by a Java class are currently not mapped to corresponding OData constructs - please log an issue if you need that functionality. OData does not have an out parameter concept, thus OUT parameters are ignored, and INOUT parameters are treated only as IN. A result set is mapped to a complex type collection result. An array result will be mapped to a simple type collection.

An OData Function will be used if:

- The procedure/function has a return value - either scalar or a result set.
- The procedure/function has no LOB input parameters - currently Clob, Blob, XML, Geometry, Geography, and JSON are considered LOB types.
- The procedure/function is side effect free - this is determined by an explicit value of 0 for the update count. For example: CREATE VIRTUAL PROCEDURE … OPTIONS (UPDATECOUNT 0) AS BEGIN …

If any one of those conditions are not met the procedure/function is represented instead by an OData Action. However if there is a result set that has a LOB value, then the procedure is not mapped at all as multiple streaming values cannot be returned.

Note that OData Functions and Actions are called differently. A Function is called by a GET request where the parameter values are included in the URI. An Action is called by a POST where the content provides the parameter values.

Currently only unbounded Functions and Actions are compatible.

You should always consult the $metadata about Functions and Actions to validate how the procedures/functions were mapped.

OpenAPI Metadata

An experimental feature is available to automatically provide a Swagger 2.0 / OpenAPI metadata via [swagger|openapi].json rather than $metadata.

Example OpenAPI 2.0 URLs

```
http://localhost:8080/odata4/northwind/NW/swagger.json
http://localhost:8080/odata4/northwind/NW/openapi.json
http://localhost:8080/odata4/northwind/NW/openapi.json?version=2
```

Example OpenAPI 3.0 URL

```
http://localhost:8080/odata4/northwind/NW/openapi.json?version=3
```

Warning	Due to all of the possible query options and expansions this metadata will be significantly larger than the OData EDM representation.

Using Teiid with Hibernate

Configuration

For the most part, interacting with Teiid VDBs (Virtual Databases) through Hibernate is no different from working with any other type of data source. First, depending on where your Hibernate application will reside, either in the same VM as the Teiid Runtime or on a separate VM, will determine which jar's are used.

- Running in same VM in the WildFly server, then the teiid-client-{version}.jar and teiid-hibernate-dialect-{version}.jar already reside in <jboss-install>/modules/org/jboss/teiid/client
- Running separate VM's,you need the Teiid JDBC Driver JAR and Teiid's Hibernate Dialect JAR in the Hibernate's classpath. The Hibernate JAR can be found in <jboss-install>/modules/org/jboss/teiid/client, teiid-hibernate-dialect-{version}.jar and the Teiid JDBC Driver JAR needs to be downloaded.

These JAR files have the `org.teiid.dialect.TeiidDialect` and `org.teiid.jdbc.TeiidDriver` and `org.teiid.jdbc.TeiidDataSource` classes.

You configure Hibernate (via hibernate.cfg.xml) as follows:

- Specify the Teiid driver class in the `connection.driver_class` property:

```
<property name="connection.driver_class">
     org.teiid.jdbc.TeiidDriver
</property>
```

- Specify the URL for the VDB in the `connection.url` property (replacing terms in angle brackets with the appropriate values):

```
<property name="connection.url">
    jdbc:teiid:<vdb-name>@mm://<host>:<port>;user=<user-name>;password=<password>
</property>
```

Tip	Be sure to use a Local JDBC Connection if Hibernate is in the same VM as the application server.

- Specify the Teiid dialect class in the `dialect` property:

```
<property name="dialect">
    org.teiid.dialect.TeiidDialect
</property>
```

Alternatively, if you put your connection properties in `hibernate.properties` instead of `hibernate.cfg.xml`, they would look like this:

```
hibernate.connection.driver_class=org.teiid.jdbc.TeiidDriver
hibernate.connection.url=jdbc:teiid:<vdb-name>@mm://<host>:<port>
hibernate.connection.username=<user-name>
hibernate.connection.password=<password>
hibernate.dialect=org.teiid.dialect.TeiidDialect
```

Note also that since your VDBs will likely contain multiple source and view models with identical table names, you will need to fully qualify table names specified in Hibernate mapping files:

```
<class name="<Class name>" table="<Source/view model name>.[<schema name>.]<Table name>">
    ...
</class>
```

Example Mapping

```
<class name="org.teiid.example.Publisher" table="BOOKS.BOOKS.PUBLISHERS">
    ...
</class>
```

Identifier Generation

SEQUENCE Based Identity Generation

If you want to use SEQUENCE based Identity generation with Teiid, you can do so through the TeiidDialect. When you define a JPA Entity

```
public class Customer {
        @GeneratedValue(strategy = GenerationType.SEQUENCE, generator =
"customer_generator")
        @SequenceGenerator(name="customer_generator", sequenceName =
"customer_seq")
        @Id
        Long id;
}
```

In the Teiid VDB, define a virtual function as below example. Note, "_nextval" appended to the sequence name on the name of the function.

```
CREATE VIRTUAL FUNCTION customer_seq_nextval() RETURNS long
AS
BEGIN
    -- Your code to retrieve the sequence from source database
    -- or generate one in Teiid.
END;
```

Given the above template, if for example you are working with Oracle would like to use the Oracle sequence you already defined as "customer_seq" in your Oracle database, then create View procedure in Teiid as

```
CREATE VIRTUAL FUNCTION customer_seq_nextval() RETURNS long
AS
BEGIN
    SELECT OracleDB.mySequence_nextval();
END;
```

Stating with Teiid 10.0, for some sources, including DB2, Oracle, H2, PostgreSQL, DB2, you can automatically import sequence information. For other sources you need to add source functions to represent the sequence calls. For example assuming you wanted to do this manually for Oracle, then in your OracleDB source model, create a source function:

```
CREATE FOREIGN FUNCTION mySequence_nextval() RETURNS long
```

```
OPTIONS ("teiid_rel:native-query" 'SELECT customer_seq.NEXTVAL FROM dual',
DETERMINISM 'NONDETERMINISTIC');
```

Then when the Customer entity is inserted, the sequence is used.

TABLE Based Idenity Generation

If you want use TABLE based Identity generation with Teiid, you can do so through the TeiidDialect. When you define a JPA Entity like

```
public class Customer {
        @TableGenerator(name = "customer",
                        table = "id_generator",
                        pkColumnName = "idkey",
                        valueColumnName = "idvalue",
                        pkColumnValue = "customer",
                        allocationSize = 1)
        @GeneratedValue(strategy = GenerationType.TABLE, generator = "customer")
        @Id
        Long id;

        ...
}
```

Then create a virtual table in Teiid’s view model as

```
CREATE VIEW id_generator (
        idkey string(255) NOT NULL,
        idvalue long,
        CONSTRAINT id_generatorPK PRIMARY KEY(idkey)
) OPTIONS (UPDATABLE TRUE)
AS
SELECT IDKEY, IDVALUE FROM OracleDB.IDGENERATOR;
```

Where in OracleDB, you have a physical Table called "IDGENERATOR" and with above shown columns. When you use this technique, please make sure you have seed content like below to begin with

```
INSERT INTO IDGENERATOR(IDKEY, IDVALUE) VALUES ('customer', 100);
```

such that the IDKEY matches and IDVALUE has a initializer value.

IDENTITY Based identity generation

- Teiid provides for GUID and Identity (using generated key retrieval) identifier generation strategy.

Limitations

- Many Hibernate use cases assume a data source has the ability (with proper user permissions) to process Data Definition Language (DDL) statements like CREATE TABLE and DROP TABLE as well as Data Manipulation Language (DML) statements like SELECT, UPDATE, INSERT and DELETE. Teiid can handle a broad range of DML, but does not directly handle DDL against a particular source.
- Sequence generation is not directly supported.

Using Teiid with EclipseLink

Overview

We can use Teiid with Hibernate, we also have a quick start to show how: Hibernate on top of Teiid. Both Hibernate and Eclipselink fully support JSR-317 (JPA 2.0). The primary purpose of this document is demonstrate how use Teiid with EclipseLink.

Configuration

For the most part, interacting with Teiid VDBs (Virtual Databases) through Eclipselink is no different from working with any other type of data source. First, depending on where your Eclipselink application will reside, either in the same VM as the Teiid Runtime or on a separate VM, will determine which jar's are used.

- Running in same VM in the WildFly server, the teiid-client-{version}.jar and teiid-eclipselink-platform-{version}.jar are needed
- Running separate VM's,you need the Teiid JDBC Driver JAR(Download Teiid JDBC Driver JAR) and Teiid's Eclipselink Platform JAR(teiid-eclipselink-platform {version}.jar) in the Eclipselink's classpath.

These JAR files have the *org.teiid.eclipselin.platform.TeiidPlatform* and *org.teiid.jdbc.TeiidDriver* classes.

You configure EclipseLink (via persistence.xml) as follows:

- Specify the Teiid driver class, connection url

```
<property name="javax.persistence.jdbc.driver" value="org.teiid.jdbc.TeiidDriver" />
<property name="javax.persistence.jdbc.url" value="jdbc:teiid:<vdb-name>@mm://<host>:<port>" />
<property name="javax.persistence.jdbc.user" value="<username>" />
<property name="javax.persistence.jdbc.password" value="<password>" />
```

- Specify the Teiid platform class

```
<property name="eclipselink.target-database" value="org.teiid.eclipselink.platform.TeiidPlatform"/>
```

Limitations

- Many Eclipselink use cases assume a data source has the ability (with proper user permissions) to process Data Definition Language (DDL) statements like CREATE TABLE and DROP TABLE as well as Data Manipulation Language (DML) statements like SELECT, UPDATE, INSERT and DELETE. Teiid can handle a broad range of DML, but does not directly support DDL against a particular source.
- Sequence generation is not directly supported.

GeoServer Integration

GeoServer is an open source server for geospatial data. It can be integrated with Teiid to serve geospatial data from a variety of sources.

Prerequisites

- Have GeoServer installed. By default this will be in a different container than the Teiid WildFly instance, but it should be possible to deploy into the same WildFly instance. Teiid integration was initially tested with GeoServer version 2.6.x, and is compatible with versions 2.8.x and 2.12.x. See TEIID-5236
- Your Teiid installation should already be setup for pg/ODBC access. This allows the built-in compatibility with GeoServer for PostGIS/PostgreSQL to be used.
- Have a VDB deployed that exposes one or more tables containing an appropriate Geometry column.
    a. The Teiid system table GEOMETY_COLUMNS will be used by GeoServer. Please ensure that the relevant geometry columns have the appropriate srid and coord_dimensions, which may require setting the {http://www.teiid.org/translator/spatial/2015}srid and {http://www.teiid.org/translator/spatial/2015}coord_dimension extension property on the geometry column.

GeoServer Configuration

This process will need to be repeated for each VDB schema you are exposing that contains geospatial data.

1. Using the GeoServer admin web application, select Stores → Add new Store. Under Vector Data Sources, select PostGIS.
2. Using the non-JNDI connection, fill in the Teiid server host, ODBC port, database (VDB Name with optional version), user, and password, schema (schema/model from the target VDB).
    i. If your VDBs contain target schema or table names with % or _, Teiid must be configured to use the same default like escape character '\' as PostgreSQL to properly respond to metadata queries. Either the system property org.teiid.backslashDefaultMatchEscape must be set to true or the Teiid session variable backslashDefaultMatchEscape must be set to true - for example enter "select cast(teiid_session_set('backslashDefaultMatchEscape', true) as boolean)" in the "Session startup SQL" to configure just this GeoServer connection pool.
3. Follow the typical GeoServer instructions for creating a Layer based upon the Teiid store.
    i. Note that Teiid is not compatible with the PostGIS function ST_Estimated_Extent and attempts to compute the bounding box from the data, result in log errors.

Additional Considerations

- If you are integrating a PostgreSQL source, you must not re-expose the geometry_columns or geography_columns tables. This is because GeoServer makes unqualified queries that reference geometry_columns and the query should resolve against the Teiid system table instead.
- Teiid does not by default expose a GT_PK_METADATA, which is optionally used by GeoServer

QGIS Integration

QGIS is an open source geospatial platform. It can be integrated with Teiid to serve geospatial data from a variety of sources.

Prerequisites

- Have QGIS installed. Teiid integration was last tested with version 2.14.
- Your Teiid installation should already be setup for ODBC access. This allows the built-in compatibility of QGIS for PostGIS/PostgreSQL to be used.
- Have a VDB deployed that exposes one or more tables containing an appropriate Geometry column.
    a. The Teiid system table GEOMETY_COLUMNS will be used by QGIS. Please ensure that the relevant geometry columns have the appropriate srid and coord_dimensions, which may require setting the {http://www.teiid.org/translator/spatial/2015}srid and {http://www.teiid.org/translator/spatial/2015}coord_dimension extension property on the geometry column.

QGIS Configuration

This process will need to be repeated for each VDB schema you are exposing that contains geospatial data.

1. In the QGIS GUI browser panel right click on PostGIS and select "New Connection".
2. Fill in the Teiid server host, ODBC port, database (VDB Name with optional version), user, and password.
    i. If your VDBs contain target schema or table names with % or _, Teiid must be configured to use the same default like escape character '\' as PostgreSQL to properly respond to metadata queries. Either the system property org.teiid.backslashDefaultMatchEscape must be set to true.
3. Follow the typical QGIS instructions for creating a Layer by browsing to the appropriate schema and selecting a table that exposes a geometry.

Additional Considerations

- If you are integrating a PostgreSQL source, you must not re-expose the postgres system tables including the PostGIS geometry_columns or geography_columns tables. This is because QGIS makes unqualified references to these tables, which may then be ambiguous.
- Operations involving creating or deleting schemas or tables will not work.
- The logs might contain messages related to information_schema.tables - this is to determine if the qgis_editor_widget_styles table exists. Teiid is not compatible with QGIS editor widget styles.

SQLAlchemy Integration

SQLAlchemy is an open source SQL toolkit and ORM for Python.

Prerequisites

- Have SQLAlchemy installed installed. Teiid integration was last tested with version 1.1.6.
- Your Teiid installation should already be setup for ODBC access. This allows the built-in compatibility with SQLAlchemy for PostgreSQL to be used.

Usage

You should be able to use a SQLAlchemy engine for querying. Reflective import of most table metadata is also provided.

Sample Usage

```
import sqlalchemy
from sqlalchemy import create_engine, Table, MetaData
engine = create_engine("postgresql+psycopg2://user:password@host:35432/vdb")
engine.connect()
#engine is ready for queries
result = connection.execute("select * from some_table")
#reflective table import
meta = MetaData()
test = Table('public.test', meta, autoload=True, autoload_with=engine,postgresql_ignore_search_path=True)
```

Limitations

Only a subset of the PostgreSQL dialect is available. The primary intent is to allow querying through Teiid. If there are additional features that are needed, please log an enhancement request.

Column metadata will not be available for tables that contain the period '.' character. Depending upon your needs, you may need import settings that use simple Teiid names and not source schema qualified names.

Application compatibility

Superset

Superset is an open source data visualization and dashboard builder. It uses SQLAlchemy to access relational sources.

Once you have followed the above instructions, you may access a Teiid VDB by adding a Database under the Sources menu.

The URL will be of the same form shown in the SQLAlchemy integration:
postgresql+psycopg2://user:password@host:35432/vdb

Basic usage scenarios involving aggregation and all basic types have been tested. If there are additional features that are needed, please log an enhancement request

Node.js Integration

Node.js is an open source event driven runtime that can be integrated with Teiid.

Prerequisites

- Have Node.js installed. The npm pagckage pg is also required. Use "
- Your Teiid installation should already be setup for ODBC access. This allows the optional compatibility with Node.js for PostGIS/PostgreSQL to be used.

Usage

For example if you have VDB called "northwind" deployed on your Teiid server, and it has table called "customers" and you are using default configuration such as

user = 'user' password = 'user' host = 127.0.0.1 port = 35432

Simple Access Example

```
const { Client } = require('pg')
const client = new Client({
    user: 'user',
    host: 'localhost',
    database: 'northwind',
    password: 'secretpassword',
    port: 35432,
})
client.connect()

client.query('SELECT CustomerID, ContactName, ContactTitle FROM Customers', (err, res) => {
    console.log(err, res)
    client.end()
})
```

Note	you do not have to programmatically specify the connection information in the code as it can be obtained from environment variables and other mechanisms - see https://node-postgres.com

For more information please refer to: https://npmjs.org/package/pg

ADO.Net Integration

Npgsql is an open source ADO.NET Data Provider for PostgreSQL. It can be integrated with Teiid to provide access from programs written in C#, Visual Basic, F#.

Prerequisites

- Install the Npgsql using the .msi Windows installer. Teiid integration was last tested with version 3.2.6.
- Your Teiid installation should already be setup for pg/ODBC access.
- Have a VDB deployed.

Npgsql Configuration

For information about the available connection parameters, see the Npgsql documentation. Not all configuration parameters have been tested for use with Teiid.

Known Limitations

- TEIID-5220 prevents displaying the metadata of tables and views, but does not affect querying. Certain tools, such as PowerBi, may have options to turn of the need to perform metadata introspection.

Reauthentication

Teiid allows for connections to be reauthenticated so that the identity on the connection can be changed rather than creating a whole new connection. If using JDBC, see the changeUser Connection extension. If using ODBC, or simply need a statement based mechanism for reauthentication, see also the SET Statement for SESSION AUTHORIZATION.

Execution Properties

Execution properties may be set on a per statement basis through the `TeiidStatement` interface or on the connection via the SET Statement. For convenience, the property keys are defined by constants on the `org.teiid.jdbc.ExecutionProperties` interface.

Table 1. **Execution Properties**

Property Name/String Constant	Description
`PROP_TXN_AUTO_WRAP / autoCommitTxn`	Same as the connection property.
`PROP_PARTIAL_RESULTS_MODE / partialResultsMode`	See the Partial Results Mode
`RESULT_SET_CACHE_MODE / resultSetCacheMode`	Same as the connection property.
`SQL_OPTION_SHOWPLAN / SHOWPLAN`	Same as the connection property.
`NOEXEC / NOEXEC`	Same as the connection property.
`JDBC4COLUMNNAMEANDLABELSEMANTICS / useJDBC4ColumnNameAndLabelSemantics`	Same as the connection property.

SET Statement

Execution properties may also be set on the connection by using the SET statement. The SET statement is not yet a language feature of Teiid and is handled only in the JDBC client. Since a JDBC clients backs the pg/ODBC transport, it will work there as well.

SET Syntax:

- SET [PAYLOAD] (parameter|SESSION AUTHORIZATION) value
- SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL (READ UNCOMMITTED|READ COMMITTED|REPEATABLE READ|SERIALIZABLE)

Syntax Rules:

- The parameter must be an identifier - it can contain spaces or other special characters only if quoted.
- The value may be either a non-quoted identifier or a quoted string literal value.
- If payload is specified, e.g. "SET PAYLOAD x y", then a session scoped payload properties object will have the corresponding name value pair set. The payload object is not fully session scoped. It will be removed from the session when the XAConnection handle is closed/returned to the pool (assumes the use of TeiidDataSource). The session scoped payload is superseded by the usage of TeiidStatement.setPayload.
- Using SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL is equivalent to calling Connection.setTransactionIsolation with the corresponding level.

The SET statement is most commonly used to control planning and execution.

- SET SHOWPLAN (ON|DEBUG|OFF)
- SET NOEXEC (ON|OFF)

Enabling Plan Debug

```
Statement s = connection.createStatement();
s.execute("SET SHOWPLAN DEBUG");
...
Statement s1 = connection.createStatement();
ResultSet rs = s1.executeQuery("select col from table");

ResultSet planRs = s1.exeuteQuery("SHOW PLAN");
planRs.next();
String debugLog = planRs.getString("DEBUG_LOG");
```

Query Plan without executing the query

```
s.execute("SET NOEXEC ON");
s.execute("SET SHOWPLAN DEBUG");
...
e.execute("SET NOEXEC OFF");
```

The SET statement may also be used to control authorization. A SET SESSION AUTHORIZATION statement will perform a Reauthentication given the credentials currently set on the connection. The connection credentials may be changed by issuing a SET PASSWORD statement. A SET PASSWORD statement does not perform a reauthentication.

Changing Session Authorization

```
Statement s = connection.createStatement();
s.execute("SET PASSWORD 'someval'");
```

```
s.execute("SET SESSION AUTHORIZATION 'newuser'");
```

SHOW Statement

The SHOW statement can be used to see a varitey of information. The SHOW statement is not yet a language feature of Teiid and is handled only in the JDBC client.

SHOW Usage:

- **SHOW PLAN**- returns a resultset with a clob column PLAN_TEXT, an xml column PLAN_XML, and a clob column DEBUG_LOG with a row containing the values from the previously executed query. If SHOWPLAN is OFF or no plan is available, no rows are returned. If SHOWPLAN is not set to DEBUG, then DEBUG_LOG will return a null value.
- **SHOW ANNOTATIONS**- returns a resultset with string columns CATEGORY, PRIORITY, ANNOTATION, RESOLUTION and a row for each annotation on the previously executed query. If SHOWPLAN is OFF or no plan is available, no rows are returned.
- **SHOW** <property> - the inverse of SET, shows the property value for the given property, returns a resultset with a single string column with a name matching the property key.
- **SHOW ALL**- returns a resultset with a NAME string column and a VALUE string column with a row entry for every property value. The SHOW statement is most commonly used to retrieve the query plan, see the plan debug example.

Transactions

Teiid provides three types of transactions from a client perspective:

1. Global
2. Local
3. Request Level

All are implemented by Teiid logically as XA transactions. See the JTA specification for more on XA Transactions.

Local Transactions

A Local transaction from a client perspective affects only a single resource, but can coordinate multiple statements.

JDBC Specific

The `Connection` class uses the `autoCommit` flag to explicitly control local transactions. By default, autoCommit is set to `true`, which indicates request level or implicit transaction control.

An example of how to use local transactions by setting the autoCommit flag to false.

Local transaction control using autoCommit

```
// Set auto commit to false and start a transaction
connection.setAutoCommit(false);

try {
    // Execute multiple updates
    Statement statement = connection.createStatement();
    statement.executeUpdate("INSERT INTO Accounts (ID, Name) VALUES (10, 'Mike')");
    statement.executeUpdate("INSERT INTO Accounts (ID, Name) VALUES (15, 'John')");
    statement.close();

    // Commit the transaction
    connection.commit();

} catch(SQLException e) {
    // If an error occurs, rollback the transaction
    connection.rollback();
}
```

This example demonstrates several things:

1. Setting autoCommit flag to false. This will start a transaction bound to the connection.
2. Executing multiple updates within the context of the transaction.
3. When the statements are complete, the transaction is committed by calling commit().
4. If an error occurs, the transaction is rolled back using the rollback() method.

Any of the following operations will end a local transaction:

1. Connection.setAutoCommit(true) – if previously set to false
2. Connection.commit()
3. Connection.rollback()
4. A transaction will be rolled back automatically if it times out.

Turning Off JDBC Local Transaction Controls

In some cases, tools or frameworks above Teiid will call setAutoCommit(false), commit() and rollback() even when all access is read-only and no transactions are necessary. In the scope of a local transaction Teiid will start and attempt to commit an XA transaction, possibly complicating configuration or causing performance degradation.

In these cases, you can override the default JDBC behavior to indicate that these methods should perform no action regardless of the commands being executed. To turn off the use of local transactions, add this property to the JDBC connection URL

```
disableLocalTxn=true
```

Tip	Turning off local transactions can be dangerous and can result in inconsistent results (if reading data) or inconsistent data in data stores (if writing data). For safety, this mode should be used only if you are certain that the calling application does not need local transactions.

Transaction Statements

Transaction control statements, which are also applicable to ODBC clients, explicitly control the local transaction boundaries. The relevant statements are:

- **START TRANSACTION**- synonym for `connection.setAutoCommit(false)`
- **COMMIT**- synonym for `connection.setAutoCommit(true)`
- **ROLLBACK**- synonym for `connection.rollback()` and returning to auto commit mode.

Request Level Transactions

Request level transactions are used when the request is not in the scope of a global or local transaction, which implies "autoCommit" is "true". In a request level transaction, your application does not need to explicitly call commit or rollback, rather every command is assumed to be its own transaction that will automatically be committed or rolled back by the server.

The Teiid Server can perform updates through virtual tables. These updates might result in an update against multiple physical systems, even though the application issues the update command against a single virtual table. Often, a user might not know whether the queried tables actually update multiple sources and require a transaction.

For that reason, the Teiid Server allows your application to automatically wrap commands in transactions when necessary. Because this wrapping incurs a performance penalty for your queries, you can choose from a number of available wrapping modes to suit your environment. You need to choose between the highest degree of integrity and performance your application needs. For example, if your data sources are not transaction-compliant, you might turn the transaction wrapping off (completely) to maximize performance.

You can set your transaction wrapping to one of the following modes:

1. *ON*: This mode always wraps every command in a transaction without checking whether it is required. This is the safest mode.
2. *OFF*: This mode never automatically wraps a command in a transaction or check whether it needs to wrap a command. This mode can be dangerous as it will allow multiple source updates outside of a transaction without an error. This mode has best performance for applications that do not use updates or transactions.
3. *DETECT*: This mode assumes that the user does not know to execute multiple source updates in a transaction. The Teiid Server checks every command to see whether it is a multiple source update and wraps it in a transaction. If it is single source then uses the source level command transaction. You can set the transaction mode as a property when you establish the Connection or on a per-query basis using the execution properties. For more information on execution properties, see the section Execution Properties

Multiple Insert Batches

When issuing an INSERT with a query expression (or the deprecated SELECT INTO), multiple insert batches handled by separate source INSERTS may be processed by the Teiid server. Be sure that the sources that you target support XA or that compensating actions are taken in the event of a failure.

Using Global Transactions

Global or client XA transactions are only applicable to JDBC clients. They all the client to coordinate multiple resources in a single transaction. To take advantage of XA transactions on the client side, use the `TeiidDataSource` (or Teiid Embedded with transaction detection enabled).

When an XAConnection is used in the context of a UserTransaction in an application server, such as JBoss, WebSphere, or Weblogic, the resulting connection will already be associated with the current XA transaction. No additional client JDBC code is necessary to interact with the XA transaction.

Usage with UserTransaction

```
   UserTransaction ut = context.getUserTransaction();
   try {
      ut.begin();
      Datasource oracle = lookup(...)
      Datasource teiid = lookup(...)

      Connection c1 = oracle.getConnection();
      Connection c2 = teiid.getConnection();

      // do something with Oracle connection
      // do something with Teiid connection
      c1.close();
      c2.close();
      ut.commit();
   } catch (Exception ex) {
       ut.rollback();
   }
```

In the case that you are not running in a JEE container environment and you have your own transaction manger to co-ordinate the XA transactions, code will look some what like below.

Manual Usage of XA transactions

```
XAConnection xaConn = null;
XAResource xaRes = null;
Connection conn = null;
Statement stmt = null;

try {
  xaConn = <XADataSource instance>.getXAConnection();
  xaRes = xaConn.getXAResource();
  Xid xid = <new Xid instance>;
  conn = xaConn.getConnection();
  stmt = conn.createStatement();

  xaRes.start(xid, XAResource.TMNOFLAGS);
  stmt.executeUpdate("insert into …");
  <other statements on this connection or other resources enlisted in this transaction>
  xaRes.end(xid, XAResource.TMSUCCESS);

  if (xaRes.prepare(xid) == XAResource.XA_OK) {
    xaRes.commit(xid, false);
  }
}
catch (XAException e) {
  xaRes.rollback(xid);
}
finally {
  <clean up>
}
```

With the use of global transactions multiple Teiid XAConnections may participate in the same transaction. The Teiid JDBC XAResource "isSameRM" method returns "true" only if connections are made to the same server instance in a cluster. If the Teiid connections are to different server instances then transactional behavior may not be the same as if they were to the same cluster member. For example, if the client transaction manager uses the same XID for each connection (which it should not since isSameRM will return false), duplicate XID exceptions may arise from the same physical source accessed through different cluster members. More commonly if the client transaction manager uses a different branch identifier for each connection, issues may arise with sources that lock or isolate changes based upon branch identifiers.

Restrictions

Application Restrictions

The use of global, local, and request level transactions are all mutually exclusive. Request level transactions only apply when not in a global or local transaction. Any attempt to mix global and local transactions concurrently will result in an exception.

Enterprise Information System (EIS) compatibility

The underlying data source that represents the EIS system and the EIS system itself must support XA transactions if they want to participate in distributed XA transaction through Teiid. If source system does not support the XA, then it can not fully participate in the distributed transaction. However, the source is still eligible to participate in data integration without the XA support.

The participation in the XA transaction is automatically determined based on the source XA capability. It is user’s responsibility to make sure that they configure a XA resource when they require them to participate in distributed transaction.

Developer's Guide

This guide contains information for developers creating custom solutions with Teiid. It covers creating JEE JCA connectors with the Teiid framework, Teiid Translators, Teiid User Defined Functions (UDFs) as well as related topics.

Integrating data from a Enterprise Information System (EIS) into Teiid, is separated into two parts.

1. A Translator, which is required.
2. An optional Resource Adapter, which will typically be a JCA Resource Adapter (also called a JEE Connector)

A Translator is used to:

- Translate a Teiid-specific command into a native command
- Execute the command
- Return batches of results translated to expected Teiid types.

A Resource Adapter is used to:

- Handles all communications with individual enterprise information system (EIS), which can include databases, data feeds, flat files, etc.
- Can be a JCA Connector or any other custom connection provider. The reason Teiid recommends and uses JCA is this specification defines how one can write, package, and configure access to EIS system in consistent manner. There are also various commercial/open source software vendors already providing JCA Connectors to access a variety of back-end systems. Refer to http://java.sun.com/j2ee/connector/.
- Abstracts Translators from many common concerns, such as connection information, resource pooling, or authentication. + Given a combination of a Translator + Resource Adapter, one can connect any EIS system to Teiid for their data integration needs.

Do You Need a New Translator?

Teiid provides several translators for common enterprise information system types. If you can use one of these enterprise information systems, you do not need to develop a custom one.

Teiid offers numerous built-in translators, including:

- **JDBC Translator** - Works with many relational databases. The JDBC translator is validated against the following database systems: Oracle, Microsoft SQL Server, IBM DB2, MySQL, Postgres, Derby, Sybase, SQP-IQ, H2, and HSQL. In addition, the JDBC Translator can often be used with other 3rd-party drivers and provides a wide range of extensibility options to specialize behavior against those drivers.
- **File Translator** - Provides a procedural way to access the file system to handle text files.
- **WS Translator** - Provides procedural access to XML content using Web Services.
- **LDAP Translator** - Accesses to LDAP directory services.
- **Salesforce Translator** - Works with Salesforce interfaces.

To see a full list of available translators, see Translators

If there's not an available translator that meets your need, Teiid provides the framework for developing your own custom translator. See the Translator Development section, as it will describe how to develop, package and deploy a custom developed translator.

Do You Need a New Resource Adapter?

As mentioned above, for every Translator that needs to gather data from external source systems, it requires a resource adapter.

The following are some of resource adapters that are available to Teiid:

- *DataSource:* This is provided by the WildFly container. This is used by the JDBC Translator.
- *File:* Provides a JEE JCA based Connector to access defined directory on the file system. This is used by the File Translator
- *WS:* Provides JEE JCA Connector to invoke Web Services using WildFly Web services stack. This is used by the WS Translator
- *LDAP:* Provides JEE JCA connector to access LDAP; Used by the LDAP Translator.
- *Salesforce:* Provides JEE JCA connector to access Salesforce by invoking their Web Service interface. Used by the SalesForce Translator.

To see a full list, see Deploying VDB Dependencies

If there's not an available resource-adapter that meets your need, Teiid provides the framework for developing your own JEE JCA Connector. See the Developing JEE Connectors section, as it will describe how to develop, package and deploy a resource adapter.

Other Teiid Development

Teiid is highly extensible in other ways:

- You may add User Defined Functions. Refer to User Defined Functions.
- You may adapt logging to your needs, which is especially useful for custom audit or command logging. Refer to Custom Logging.
- You may change the subsystem for custom authentication and authorization. Refer to Custom Login Modules.

Developing JEE Connectors

Developing (Custom) JEE Connectors (Resource Adapters)

This chapter examines how to use facilities provided by the Teiid API to develop a JEE JCA Connector. Please note that these are standard JEE JCA connectors, nothing special needs to be done for Teiid. As an aid to our Translator developers, we provided a base implementation framework. If you already have a JCA Connector or some other mechanism to get data from your source system, you can skip this chapter.

If you are not familiar with JCA API, please read the JCA 1.5 Specification at http://java.sun.com/j2ee/connector/. There are lot of online tutorials on how to design and build a JCA Connector. The below are high-level steps for creating a very simple connector, however building actual connector that supports transactions, security can get much more complex.

1. Understand the JEE Connector specification to have basic idea about what JCA connectors are how they are developed and packaged. Refer to http://java.sun.com/j2ee/connector/.
2. Gather all necessary information about your Enterprise Information System (EIS). You will need to know:
    - API for accessing the system
    - Configuration and connection information for the system
    - Expectation for incoming queries/metadata
    - The processing constructs, or capabilities, supported by information system.
    - Required properties for the connection, such as URL, user name, etc.
3. Base classes for all of the required supporting JCA SPI classes are provided by the Teiid API. The JCA CCI support is not provided from Teiid, since Teiid uses the Translator API as it’s common client interface. You will want to extend:
    - BasicConnectionFactory – Defines the Connection Factory
    - BasicConnection – represents a connection to the source.
    - BasicResourceAdapter – Specifies the resource adapter class
4. Package your resource adapter. Refer to Packaging the Adapter.
5. Deploy your resource adapter. Refer to Packaging the Adapter.

For sample resource adapter code refer to the Teiid Source code at https://github.com/teiid/teiid/tree/master/connectors/.

Refer to the JBoss Application Server Connectors documentation at http://docs.jboss.org/jbossas/jboss4guide/r4/html/ch7.chapt.html.

Archetype Template Connector Project

One way to start developing a custom connector (resource-adapter) is to create a project using the Teiid archetype template. When the project is created from the template, it will contain the essential classes and resources for you to begin adding your custom logic. Additionally, the maven dependencies are defined in the pom.xml so that you can begin compiling the classes.

Note	The project will be created as an independent project and has no parent maven dependencies. It's designed to be built independent of building Teiid.

You have 2 options for creating a connector project; in Eclipse by creating a new maven project from the arche type or by using the command line to generate the project.

Create Project in Eclipse

To create a Java project in Eclipse from an arche type, perform the following:

- Open the JAVA perspective
- From the menu select File –> New —> Other
- In the tree, expand Maven and select Maven Project, press Next
- On the "Select project name and Location" window, you can accept the defaults, press Next
- On the "Select an Archetype" window, select Configure button
- Add the remote catalog: link:http://central.maven.org/maven2/ then click OK to return
- Enter "teiid" in the filter to see the Teiid arche types.
- Select the connector-archetype, then press Next
- Enter all the information (i.e., Group ID, Artifact ID, etc.) needed to generate the project, then click Finish

The project will be created and name according to the **ArtifactID**.

Create Project using Command Line

Note	make sure the link:http://central.maven.org/maven2/ repository is accessible via your maven settings.

To create a custom connector project from the command line, you can use the following template command:

- TEMPLATE

```
mvn archetype:generate       \
  -DarchetypeGroupId=org.teiid.arche-types                \
  -DarchetypeArtifactId=connector-archetype          \
  -DarchetypeVersion=${archetypeVersion}                \
  -DgroupId=${groupId}                  \
  -DartifactId=connector-${connector-type}  \
  -Dpackage=${package}    \
  -Dversion=${version}    \
  -Dconnector-type=${connector-type}   \
```

```
  -Dconnector-name=${connector-name}   \
  -Dvendor-name=${vendor-name}    \
  -Dteiid-version=${teiid-version}
```

- where:

```
-DarchetypeGroupId    -  is the group ID for the arche type to use to generate
-DarchetypeArtifactId -  is the artifact ID for the arche type to use to generate
-DarchetypeVersion    -  is the version for the arche type to use to generate
-DgroupId     -  (user defined) group ID for the new connector project pom.xml
-DartifactId      -  (user defined) artifact ID for the new connector project
pom.xml
-Dpackage     -  (user defined) the package structure where the java and resource
files will be created
-Dversion     -  (user defined) the version that the new connector project pom.xml
will be
-Dconnector-type  -  (user defined) the type of the new connector project, used in
defining the package name
-Dconnector-name  -  (user defined) the name of the new connector project, used as
the prefix to creating the java class names
-Dvendor-name     -  name of the Vendor for the data source, updates the rar
-Dteiid-version   -  the Teiid version the connector will depend upon
```

- EXAMPLE
- this is an example of the template that can be run:

```
mvn archetype:generate                    \
  -DarchetypeGroupId=org.teiid.arche-types   \
  -DarchetypeArtifactId=connector-archetype  \
  -DarchetypeVersion=12.0.0  \
  -DgroupId=org.example  \
  -Dpackage=org.example.adapter.type \
  -DartifactId=adapter-type \
  -Dversion=0.0.1-SNAPSHOT    \
  -Dconnector-type=type   \
  -Dconnector-name=Type   \
  -Dvendor-name=Vendor  \
  -Dteiid-version=14.0.0
```

When executed, you will be asked to confirm the package property

Confirm properties configuration: groupId: org.example artifactId: adapter-type version: 0.0.1-SNAPSHOT package: org.example.adapter.type connector-type: type connector-name: Type vendor-name: Vendor teiid-version: 14.0.0 Y: :

type Y (yes) and press enter, and the creation of the connector project will be done

Upon creation, a directory based on the **artifactId* will be created, that will contain the project. Note: The project will not compile because the ${connector-name} Connection interface in the ConnectionImpl has not been added as a dependency in the pom.xml. This will need to be done.

Now you are ready to start adding your custom code.

Implementing the Teiid Framework

If you are going to use the Teiid framework for developing a JCA connector, follow these steps. The required classes are in `org.teiid.resource.api` package. Please note that Teiid framework does not make use JCA's CCI framework, only the JCA's SPI interfaces.

- Define Managed Connection Factory
- Define the Connection Factory class
- Define the Connection class
- Define the configuration properties in a "ra.xml" file

Define Managed Connection Factory

Extend the `BasicManagedConnectionFactory` , and provide a implementation for the "createConnectionFactory()" method. This method defines a factory method that can create connections.

This class also defines configuration variables, like user, password, URL etc to connect to the EIS system. Define an attribute for each configuration variable, and then provide both "getter" and "setter" methods for them. Note to use only "java.lang" objects as the attributes, DO NOT use Java primitives for defining and accessing the properties. See the following code for an example.

```
public class MyManagedConnectionFactory extends BasicManagedConnectionFactory
{
   @Override
   public Object createConnectionFactory() throws ResourceException
   {
      return new MyConnectionFactory();
   }

   // config property name (metadata for these are defined inside the ra.xml)
   String userName;
   public String getUserName()          {  return this.userName;  }
   public void setUserName(String name){  this.userName = name;  }

   // config property count  (metadata for these are defined inside the ra.xml)
   Integer count;
   public Integer getCount()            {  return this.count;  }
   public void setCount(Integer value) {  this.count = value; }

}
```

Define the Connection Factory class

Extend the `BasicConnectionFactory` class, and provide a implementation for the "getConnection()" method.

```
public class MyConnectionFactory extends BasicConnectionFactory
{
   @Override
   public MyConnection getConnection() throws ResourceException
   {
      return new MyConnection();
   }
}
```

Since the Managed connection object created the "ConnectionFactory" class it has access to all the configuration parameters, if "getConnection" method needs to do pass any of credentials to the underlying EIS system. The Connection Factory class can also get reference to the calling user's `javax.security.auth.Subject` during "getConnection" method by calling

```
Subject subject = ConnectionContext.getSubject();
```

This "Subject" object can give access to logged-in user's credentials and roles that are defined. Note that this may be null.

Note that you can define "security-domain" for this resource adapter, that is separate from the Teiid defined "security-domain" for validating the JDBC end user. However, it is the user's responsibility to make the necessary logins before the Container's thread accesses this resource adapter, and this can get overly complex.

Define the Connection class

Extend the `BasicConnection` class, and provide a implementation based on your access of the Connection object in the Translator. If your connection is stateful, then override "isAlive()" and "cleanup()" methods and provide proper implementations. These are called to check if a Connection is stale or need to flush them from the connection pool etc. by the Container.

```
public class MyConnection extends BasicConnection
{
   public void doSomeOperation(command)
   {
      // do some operation with EIS system..
      // This is method you use in the Translator, you should know
      // what need to be done here for your source..
   }

   @Override
   public boolean isAlive()
   {
      return true;
   }

   @Override
   public void cleanUp()
   {

   }
}
```

XA Transactions

If your EIS source can participate in XA transactions, then on your Connection object, override the "getXAResource()" method and provide the "XAResource" object for the EIS system. Refer to Define the Connection class. Also, You need to extend the "BasicResourceAdapter" class and provide implementation for method "public XAResource[] getXAResources(ActivationSpec[] specs)" to participate in crash recovery.

Note that, only when the resource adapters are XA capable, then Teiid can make them participate in a distributed transactions. If they are not XA capable, then source can participate in distributed query but will not participate in the transaction. Transaction semantics are defined by how you you configured "connection-factory" in a "resource-adapter". i.e. jta=true/false.

Define the configuration properties in a "ra.xml" file

Define a "ra.xml" file in "META-INF" directory of your RAR file. An example file is provided in ra.xml file Template.

For every attribute defined inside the your ManagedConnectionFactory class, define the following XML configuration for that attribute inside the "ra.xml" file. These properties are used by user to configure instance of this Connector inside a Container. Also, during the startup the Container reads these properties from this file and knows how to inject provided values in the datasource definition into an instance of "ManagedConnectionFactory" to create the Connection. Refer to Developing JEE Connectors#Define Managed Connection Factory.

```
<config-property>
   <description>
      {$display:"${display-name}",$description:"${description}", $allowed="${allowed}",
      $required="${true|false}", $defaultValue="${default-value}"}
   </description>
   <config-property-name>${property-name}</config-property-name>
   <config-property-type>${property-type}</config-property-type>
   <config-property-value>${optioal-property-value}</config-property-value>
</config-property>
```

The format and contents of "<description>" element may be used as extended metadata for tooling. The special format must begin and end with curly braces e.g. {…}. This use of the special format and all properties is optional. Property names begin with '$' and are separated from the value with ':'. Double quotes identifies a single value. A pair of square brackets, e.g. […], containing comma separated double quoted entries denotes a list value.

Extended metadata properties

- $display: Display name of the property
- $description: Description about the property
- $required: The property is a required property; or optional and a default is supplied
- $allowed: If property value must be in certain set of legal values, this defines all the allowed values
- $masked: The tools need to mask the property; Do not show in plain text; used for passwords
- $advanced: Notes this as Advanced property
- $editable: Property can be modified; or read-only

Note that all these are optional properties; however in the absence of this metadata, Teiid tooling may not work as expected.

ra.xml file Template

This appendix contains an example of the ra.xml file that can be used as a template when creating a new Connector.

```
<?xml version="1.0" encoding="UTF-8"?>
<connector xmlns="http://java.sun.com/xml/ns/j2ee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
   http://java.sun.com/xml/ns/j2ee/connector_1_5.xsd" version="1.5">

   <vendor-name>${comapany-name}</vendor-name>
   <eis-type>${type-of-connector}</eis-type>
   <resourceadapter-version>1.0</resourceadapter-version>
   <license>
      <description>${license text}</description>
      <license-required>true</license-required>
   </license>

   <resourceadapter>
      <resourceadapter-class>org.teiid.resource.spi.BasicResourceAdapter</resourceadapter-class>
      <outbound-resourceadapter>
         <connection-definition>
            <managedconnectionfactory-class>${connection-factory}</managedconnectionfactory-class>

            <!-- repeat for every configuration property -->
            <config-property>
               <description>
                  {$display:"${short-name}",$description:"${description}",$allowed:[${value-list}],
                  $required:"${required-boolean}", $defaultValue:"${default-value}"}
               </description>
               <config-property-name>${property-name}</config-property-name>
               <config-property-type>${property-type}</config-property-type>
               <config-property-value>${optional-property-value}</config-property-value>
            </config-property>

            <!-- use the below as is if you used the Connection Factory interface -->
            <connectionfactory-interface>
               javax.resource.cci.ConnectionFactory
            </connectionfactory-interface>

            <connectionfactory-impl-class>
               org.teiid.resource.spi.WrappedConnectionFactory
            </connectionfactory-impl-class>

            <connection-interface>
               javax.resource.cci.Connection
            </connection-interface>

            <connection-impl-class>
               org.teiid.resource.spi.WrappedConnection
            </connection-impl-class>

         </connection-definition>

         <transaction-support>NoTransaction</transaction-support>

         <authentication-mechanism>
            <authentication-mechanism-type>BasicPassword</authentication-mechanism-type>
            <credential-interface>
               javax.resource.spi.security.PasswordCredential
            </credential-interface>
         </authentication-mechanism>
         <reauthentication-support>false</reauthentication-support>

      </outbound-resourceadapter>
```

```
    </resourceadapter>

</connector>
```

${…} indicates a value to be supplied by the developer.

Packaging the Adapter

Once all the required code is developed, it is time to package them into a RAR artifact, that can be deployed into a Container. A RAR artifact is similar to a WAR. To put together a RAR file it really depends upon the build system you are using.

- Eclipse: You can start out with building Java Connector project, it will produce the RAR file
- Ant: If you are using "ant" build tool, there is "rar" build task available
- Maven: If you are using maven, use <packaging> element value as "rar". Teiid uses maven, you can look at any of the "connector" projects for sample "pom.xml" file. See `Build Environment` for an example of a pom.xml file.

Make sure that the RAR file, under its "META-INF" directory has the "ra.xml" file. If you are using maven refer to http://maven.apache.org/plugins/maven-rar-plugin/. In the root of the RAR file, you can embed the JAR file containing your connector code and any dependent library JAR files.

Adding Dependent Libraries

Add MANIFEST.MF file in the META-INF directory, and the following line to add the core Teiid API dependencies for resource adapter.

```
Dependencies: org.jboss.teiid.common-core,org.jboss.teiid.api,javax.api
```

If your resource adapter depends upon any other third party jar files, .dll or .so files they can be placed at the root of the rar file. If any of these libraries are already available as modules in WildFly, then you can add the module name to the above MANIFEST.MF file to define as dependency.

Deploying the Adapter

Once the RAR file is built, deploy it by copying the RAR file into "deploy" directory of WildFly's chosen profile. Typically the server does not need to be restarted when a new RAR file is being added. Alternatively, you can also use "admin-console", a web based monitoring and configuration tool, to deploy this file into the container.

Once the Connector's RAR file is deployed into the WildFly container, now you can create an instance of this connector to be used with your Translator. Creating an instance of this Connector is no different than creating a "Connection Factory" in WildFly. Again, you have two ways to create a ""ConnectionFactory".

Edit standalone.xml or domain.xml file, and add following XML in the "resource-adapters" subystem.

```
<!-- If susbsytem is already defined, only copy the contents under it and edit to suit your needs -->
<subsystem xmlns="urn:jboss:domain:resource-adapters:1.0">
    <resource-adapters>
        <resource-adapter>
            <archive>teiid-connector-sample.rar</archive>
            <transaction-support>NoTransaction</transaction-support>
            <connection-definitions>
                <connection-definition class-name="org.teiid.resource.adapter.MyManagedConnectionFactory"  jndi
-name="${jndi-name}"
                        enabled="true"
                        use-java-context="true"
                        pool-name="sample-ds">
                      <config-property name="UserName">jdoe</config-property>
                      <config-property name="Count">12</config-property>
                </connection-definition>
            </connection-definitions>
        </resource-adapter>
    </resource-adapters>
</subsystem>
```

There are lot more properties that you can define for pooling, transactions, security, etc., in this file. Check the WildFly documentation for all the available properties.

Alternatively, you can use the web based ""admin-console" configuration and monitoring program, to create a new Connection Factory. Have your RAR file name and needed configuration properties handy and fill out web form to create the ConnectionFactory.

Translator (Custom) Development

Below are the high-level steps for creating custom Translators, which is described in this section. This section will cover how to do each of the following steps in detail. It also provides additional information for advanced topics, such as streaming large objects.

For sample Translator code, refer to the Teiid source code at https://github.com/teiid/teiid/tree/master/connectors/.

1. Create a new or reuse an existing Resource Adapter for the EIS system, to be used with this Translator. Refer to Custom Resource Adapters.
2. Decide whether to use the Teiid archetype template to create your initial custom translator project and classes or manually create your environment. Refer to Environment Setup.
3. Implement the required classes defined by the Translator API. Refer to Implementing the Framework.
   1) Create an ExecutionFactory – Extend the `org.teiid.translator.ExecutionFactory` class 2) Create relevant Executions (and sub-interfaces) – specifies how to execute each type of command
4. Define the template for exposing configuration properties. Refer to Packaging.
5. Deploy your Translator. Refer to Deployment.
6. Deploy a Virtual Database (VDB) that uses your Translator.
7. Execute queries via Teiid.

Translator Environment Setup

To setup the environment for developing a custom translator, you have 2 options;

1. Manually setup the build environment - structure, framework classes, and resources.
2. Use the Teiid Translator Archetype template to generate the initial project.

Setting up the build environment

For Eclipse users (without maven integration), create a java project and add dependencies to "teiid-common-core", "teiid-api" and JEE "connector-api" jars.

For maven users add the following as your dependencies:

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.teiid</groupId>
            <artifactId>teiid-bom</artifactId>
            <version>${teiid-version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
<dependencies>
    <dependency>
        <groupId>org.teiid</groupId>
        <artifactId>teiid-api</artifactId>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>org.teiid</groupId>
        <artifactId>teiid-resource-spi</artifactId>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>org.teiid</groupId>
        <artifactId>teiid-common-core</artifactId>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>javax.resource</groupId>
        <artifactId>connector-api</artifactId>
        <scope>provided</scope>
    </dependency>
</dependencies>
```

Where the ${teiid-version} property should be set to the expected version, such as 14.0.0. You can find Teiid artifacts in the Maven Central Repository.

Archetype Template Translator Project

One way to start developing a custom translator is to create a project using the Teiid archetype template. When the project is created from the template, it will contain the essential classes (i.e., ExecutionFactory) and resources for you to begin adding your custom logic. Additionally, the maven dependencies are defined in the pom.xml so that you can begin compiling the classes.

Note	The project will be created as an independent project and has no parent maven dependencies. It's designed to be built independent of building Teiid.

You have 2 options for creating a translator project; in Eclipse by creating a new maven project from the arche type or by using the command line to generate the project.

Create Project in Eclipse

To create a Java project in Eclipse from an arche type, perform the following:

- Open the JAVA perspective
- From the menu select File –> New —> Other
- In the tree, expand Maven and select Maven Project, press Next
- On the "Select project name and Location" window, you can accept the defaults, press Next
- On the "Select an Archetype" window, select Configure button
- Add the remote catalog: link:http://central.maven.org/maven2/ then click OK to return
- Enter "teiid" in the filter to see the Teiid arche types.
- Select the translator-archetype v12.0.0, then press Next
- Enter all the information (i.e., Group ID, Artifact ID, etc.) needed to generate the project, then click Finish

The project will be created and name according to the **ArtifactID**.

Create Project using Command Line

Note	make sure the link:http://central.maven.org/maven2/ repository is accessible via your maven settings.

To create a custom translator project from the command line, you can use the following template command:

- TEMPLATE

```
mvn archetype:generate                \
  -DarchetypeGroupId=org.teiid.arche-types               \
  -DarchetypeArtifactId=translator-archetype          \
  -DarchetypeVersion=${archetypeVersion}               \
  -DgroupId=${groupId}                  \
  -DartifactId=translator-${translator-type}    \
  -Dpackage=${package}    \
  -Dversion=${version}    \
  -Dtranslator-type=${translator-type}   \
  -Dtranslator-name=${translator-name}
  -Dteiid-version=${teiid-version}
```

- where:

```
  -DarchetypeGroupId    -  is the group ID for the arche type to use to generate
  -DarchetypeArtifactId -  is the artifact ID for the arche type to use to generate
  -DarchetypeVersion    -  is the version for the arche type to use to generate
  -DgroupId     -  (user defined) group ID for the new translator project pom.xml
  -DartifactId      -  (user defined) artifact ID for the new translator project pom.xml
  -Dpackage     -  (user defined) the package structure where the java and resource files will be created
  -Dversion     -  (user defined) the version that the new connector project pom.xml will be
  -Dtranslator-type -  (user defined) the translator type that's used by Teiid when mapping the physical source
 to the translator to use
  -Dtranslator-name -  (user defined) the translator name thats used for name the java class names
  -Dteiid-version   -  the Teiid version the connector will depend upon
```

- EXAMPLE
- this is an example of the template that can be run:

```
mvn archetype:generate                    \
  -DarchetypeGroupId=org.teiid.arche-types   \
  -DarchetypeArtifactId=translator-archetype  \
  -DarchetypeVersion=  \
  -DgroupId=org.example  \
  -DartifactId=translator-type  \
  -Dpackage=org.example.translator.type    \
  -Dversion=0.0.1-SNAPSHOT  \
  -Dtranslator-type=type  \
  -Dtranslator-name=Type  \
  -Dteiid-version=14.0.0
```

When executed, you will be asked to confirm the properties

Confirm properties configuration: groupId: org.example artifactId: translator-type version: 0.0.1-SNAPSHOT package: org.example.translator.type teiid-version: 14.0.0 translator-name: Type translator-type: type Y: : y

type Y (yes) and press enter, and the creation of the translator project will be done

Upon creation, a directory based on the **artifactId** will be created, that will contain the project. 'cd' into that directory and execute a test build to confirm the project was created correctly:

```
mvn clean install
```

This should build successfully, and now you are ready to start adding your custom code.

Implementing the Framework

Caching API

Translators may contribute cache entries to the result set cache by the use of the `CacheDirective` object. Translators wishing to participate in caching should return a `CacheDirective` from the `ExecutionFactory.getCacheDirective` method, which is called prior to execution. The command passed to `getCacheDirective` will already have been vetted to ensure that the results are eligible for caching. For example update commands or commands with pushed dependent sets will not be eligible for caching.

If the translator returns null for the `CacheDirective` , which is the default implementation, the engine will not cache the translator results beyond the current command. It is up to your custom translator or custom delegating translator to implement your desired caching policy.

Note	In special circumstances where the translator has performed it's own caching, it can indicate to the engine that the results should not be cached or reused by setting the `Scope` to `Scope.NONE` .

The returned `CacheDirective` will be set on the `ExecutionContext` and is available via the `ExecutionContext.getCacheDirective()` method. Having `ExeuctionFactory.getCacheDirective` called prior to execution allows the translator to potentially be selective about which results to even attempt to cache. Since there is a resource overhead with creating and storing the cached results it may not be desirable to attempt to cache all results if it's possible to return large results that have a low usage factor. If you are unsure about whether to cache a particular command result you may return an initial `CacheDirective` then change the `Scope` to `Scope.NONE` at any time prior to the final cache entry being created and the engine will give up creating the entry and release it's resources.

If you plan on modifying the `CacheDirective` during execution, just make sure to return a new instance from the `ExecutionFactory.getCacheDirective` call, rather than returning a shared instance.

The `CacheDirective` readAll Boolean field is used to control whether the entire result should be read if not all of the results were consumed by the engine. If readAll is false then any partial usage of the result will not result in it being added as a cache entry. Partial use is determined after any implicit or explicit limit has been applied. The other fields on the `CacheDirective` object map to the cache hint options. See the table below for the default values for all options.

option	default
scope	Session
ttl	rs cache ttl
readAll	true
updatable	true
prefersMemory	false

Command Language

Language

Teiid sends commands to your Translator in object form. These classes are all defined in the "org.teiid.language" package. These objects can be combined to represent any possible command that Teiid may send to the Translator. However, it is possible to notify Teiid that your Translator can only accept certain kinds of constructs via the capabilities defined on the "ExecutionFactory" class. Refer to Translator Capabilities for more information.

The language objects all extend from the `LanguageObject` interface. Language objects should be thought of as a tree where each node is a language object that has zero or more child language objects of types that are dependent on the current node.

All commands sent to your Translator are in the form of these language trees, where the root of the tree is a subclass of `Command`. Command has several sub-classes, namely:

- `QueryExpression`
- `Insert` - also represents an upsert, see the isUpsert flag.
- `Update`
- `Delete`
- `BatchedUpdates`
- `Call`

Important components of these commands are expressions, criteria, and joins, which are examined in closer detail below. For more on the classes and interfaces described here, refer to the Teiid JavaDocs http://docs.jboss.org/teiid/7.6/apidocs.

Expressions

An expression represents a single value in context, although in some cases that value may change as the query is evaluated. For example, a literal value, such as 5 represents an integer value. An column reference such as "table.EmployeeName" represents a column in a data source and may take on many values while the command is being evaluated.

- `Expression` – base expression interface
- `ColumnReference` – represents an column in the data source
- `Literal` – represents a literal scalar value.
- `Parameter` – represents a parameter with multiple values. The command should be an instance of BulkCommand, which provides all values via getParameterValues.
- `Function` – represents a scalar function with parameters that are also Expressions
- `AggregateFunction` – represents an aggregate function which can hold a single expression
- `WindowFunction` – represents an window function which holds an AggregateFunction (which is also used to represent analytical functions) and a WindowSpecification
- `ScalarSubquery` – represents a subquery that returns a single value
- `SearchedCase, SearchedWhenClause` – represents a searched CASE expression. The searched CASE expression evaluates the criteria in WHEN clauses till one evaluates to TRUE, then evaluates the associated THEN clause.

- `Array` – represents an array of expressions, currently only used by the engine in multi-attribute dependent joins - see the supportsArrayType capability.

Condition

A criteria is a combination of expressions and operators that evaluates to true, false, or unknown. Criteria are most commonly used in the WHERE or HAVING clauses.

- `Condition` – the base criteria interface
- `Not` – used to NOT another criteria
- `AndOr` – used to combine other criteria via AND or OR
- `SubuqeryComparison` – represents a comparison criteria with a subquery including a quantifier such as SOME or ALL
- `Comparison` – represents a comparison criteria with =, >, <, etc.
- `BaseInCondition` – base class for an IN criteria
- `In` – represents an IN criteria that has a set of expressions for values
- `SubqueryIn` – represents an IN criteria that uses a subquery to produce the value set
- `IsNull` – represents an IS NULL criteria
- `Exists` – represents an EXISTS criteria that determines whether a subquery will return any values
- `Like` – represents a LIKE/SIMILAR TO/LIKE_REGEX criteria that compares string values

The FROM Clause

The FROM clause contains a list of `TableReference` ’s.

- `NamedTable` – represents a single Table
- `Join` – has a left and right `TableReference` and information on the join between the items
- `DerivedTable` – represents a table defined by an inline `QueryExpression`

A list of `TableReference` are used by default, in the pushdown query when no outer joins are used. If an outer join is used anywhere in the join tree, there will be a tree of `Join` s with a single root. This latter form is the ANSI preferred style. If you wish all pushdown queries containing joins to be in ANSI style have the capability "useAnsiJoin" return true. Refer to Command Form for more information.

QueryExpression Structure

`QueryExpression` is the base for both SELECT queries and set queries. It may optionally take an `OrderBy` (representing a SQL ORDER BY clause), a `Limit` (represent a SQL LIMIT clause), or a `With` (represents a SQL WITH clause).

Select Structure

Each `QueryExpression` can be a `Select` describing the expressions (typically ``ColumnReference`s) being selected and an `TableReference`` specifying the table or tables being selected from, along with any join information. The `Select` may optionally also supply an `Condition` (representing a SQL WHERE clause), a `GroupBy` (representing a SQL GROUP BY clause), an an `Condition` (representing a SQL HAVING clause).

SetQuery Structure

A `QueryExpression` can also be a `SetQuery` that represents on of the SQL set operations (UNION, INTERSECT, EXCEPT) on two `QueryExpression` . The all flag may be set to indicate UNION ALL (currently INTERSECT and EXCEPT ALL are not allowed in Teiid)

With Structure

A `With` clause contains named `QueryExpressions` held by `WithItems` that can be referenced as tables in the main `QueryExpression` .

Insert Structure

Each `Insert` will have a single `NamedTable` specifying the table being inserted into. It will also has a list of `ColumnReference` specifying the columns of the `NamedTable` that are being inserted into. It also has `InsertValueSource` , which will be a list of Expressions (`ExpressionValueSource`) or a `QueryExpression`

Update Structure

Each `Update` will have a single `NamedTable` specifying the table being updated and list of `SetClause` entries that specify `ColumnReference` and `Expression` pairs for the update. The Update may optionally provide a criteria `Condition` specifying which rows should be updated.

Delete Structure

Each `Delete` will have a single `NamedTable` specifying the table being deleted from. It may also optionally have a criteria specifying which rows should be deleted.

Call Structure

Each `Call` has zero or more `Argument` objects. The `Argument` objects describe the input parameters, the output result set, and the output parameters.

BatchedUpdates Structure

Each `BatchedUpdates` has a list of `Command` objects (which must be either `Insert` , `Update` or `Delete`) that compose the batch.

Language Utilities

This section covers utilities available when using, creating, and manipulating the language interfaces.

Data Types

The Translator API contains an interface `TypeFacility` that defines data types and provides value translation facilities. This interface can be obtained from calling "getTypeFacility()" method on the "ExecutionFactory" class.

The TypeFacitlity interface has methods that support data type transformation and detection of appropriate runtime or JDBC types. The TypeFacility.RUNTIME_TYPES and TypeFacility.RUNTIME_NAMES interfaces defines constants for all Teiid runtime data types. All `Expression` instances define a data type based on this set of types. These constants are often needed in understanding or creating language interfaces.

Language Manipulation

In Translators that support a fuller set of capabilities (those that generally are translating to a language of comparable to SQL), there is often a need to manipulate or create language interfaces to move closer to the syntax of choice. Some utilities are provided for this purpose:

Similar to the TypeFacility, you can call "getLanguageFactory()" method on the "ExecutionFactory" to get a reference to the `LanguageFactory` instance for your translator. This interface is a factory that can be used to create new instances of all the concrete language interface objects.

Some helpful utilities for working with `Condition` objects are provided in the `LanguageUtil` class. This class has methods to combine `Condition` with AND or to break an `Condition` apart based on AND operators. These utilities are helpful for breaking apart a criteria into individual filters that your translator can implement.

Runtime Metadata

Teiid uses a library of metadata, known as "runtime metadata" for each virtual database that is deployed in Teiid. The runtime metadata is a subset of metadata as defined by models in the Teiid models that compose the virtual database. Extension metadata may be associated via the OPTIONS clause. At runtime, using this runtime metadata interface, you get access to those set properties defined during the design time, to define/hint any execution behavior.

Translator gets access to the `RuntimeMetadata` interface at the time of `Excecution` creation. Translators can access runtime metadata by using the interfaces defined in `org.teiid.metadata` package. This package defines API representing a Schema, Table, Columns and Procedures, and ways to navigate these objects.

Metadata Objects

All the language objects extend `AbstractMetadataRecord` class

- Column - returns Column metadata record
- Table - returns a Table metadata record
- Procedure - returns a Procedure metadata record
- ProcedureParameter - returns a Procedure Parameter metadata record

Once a metadata record has been obtained, it is possible to use its metadata about that object or to find other related metadata.

Access to Runtime Metadata

The RuntimeMetadata interface is passed in for the creation of an "Execution". See "createExecution" method on the "ExecutionFactory" class. It provides the ability to look up metadata records based on their fully qualified names in the VDB.

The process of getting a Table's properties is sometimes needed for translator development. For example to get the "NameInSource" property or all extension properties:

Obtaining Metadata Properties

```
//getting the Table metadata from an Table is straight-forward
Table table = runtimeMetadata.getTable("table-name");
String contextName = table.getNameInSource();

//The props will contain extension properties
Map<String, String> props = table.getProperties();
```

Language Visitors

Framework

The API provides a language visitor framework in the `org.teiid.language.visitor` package. The framework provides utilities useful in navigating and extracting information from trees of language objects.

The visitor framework is a variant of the Visitor design pattern, which is documented in several popular design pattern references. The visitor pattern encompasses two primary operations: traversing the nodes of a graph (also known as iteration) and performing some action at each node of the graph. In this case, the nodes are language interface objects and the graph is really a tree rooted at some node. The provided framework allows for customization of both aspects of visiting.

The base `AbstractLanguageVisitor` class defines the visit methods for all leaf language interfaces that can exist in the tree. The LanguageObject interface defines an acceptVisitor() method – this method will call back on the visit method of the visitor to complete the contract. A base class with empty visit methods is provided as AbstractLanguageVisitor. The AbstractLanguageVisitor is just a visitor shell – it performs no actions when visiting nodes and does not provide any iteration.

The `HierarchyVisitor` provides the basic code for walking a language object tree. `The HierarchyVisitor` performs no action as it walks the tree – it just encapsulates the knowledge of how to walk it. If your translator wants to provide a custom iteration that walks the objects in a special order (to exclude nodes, include nodes multiple times, conditionally include nodes, etc) then you must either extend HierarchyVisitor or build your own iteration visitor. In general, that is not necessary.

The `DelegatingHierarchyVisitor` is a special subclass of the HierarchyVisitor that provides the ability to perform a different visitor's processing before and after iteration. This allows users of this class to implement either pre- or post-order processing based on the HierarchyVisitor. Two helper methods are provided on `DelegatingHierarchyVisitor` to aid in executing pre- and post-order visitors.

Provided Visitors

The `SQLStringVisitor` is a special visitor that can traverse a tree of language interfaces and output the equivalent Teiid SQL. This visitor can be used to print language objects for debugging and logging. The `SQLStringVisitor` does not use the `HierarchyVisitor` described in the last section; it provides both iteration and processing type functionality in a single custom visitor.

The `CollectorVisitor` is a handy utility to collect all language objects of a certain type in a tree. Some additional helper methods exist to do common tasks such as retrieving all `ColumnReference`s in a tree, retrieving all groups in a tree, and so on.

Writing a Visitor

Writing your own visitor can be quite easy if you use the provided facilities. If the normal method of iterating the language tree is sufficient, then just follow these steps:

Create a subclass of AbstractLanguageVisitor. Override any visit methods needed for your processing. For instance, if you wanted to count the number of `ColumnReference`s in the tree, you need only override the `visit(ColumnReference)` method. Collect any state in local variables and provide accessor methods for that state.

Decide whether to use pre-order or post-order iteration. Note that visitation order is based upon syntax ordering of SQL clauses - not processing order.

Write code to execute your visitor using the utility methods on DelegatingHierarchyVisitor:

```
// Get object tree
LanguageObject objectTree = …

// Create your visitor initialize as necessary
MyVisitor visitor = new MyVisitor();

// Call the visitor using pre-order visitation
DelegatingHierarchyVisitor.preOrderVisit(visitor, objectTree);

// Retrieve state collected while visiting
int count = visitor.getCount();
```

Connections to Source

Obtaining connections

The extended "ExecutionFactory" must implement the `getConnection()` method to allow the Connector Manager to obtain a connection.

Releasing Connections

Once the Connector Manager has obtained a connection, it will use that connection only for the lifetime of the request. When the request has completed, the closeConnection() method called on the "ExecutionFactory". You must also override this method to properly close the connection.

In cases (such as when a connection is stateful and expensive to create), connections should be pooled. If the resource adapter is JEE JCA connector based, then pooling is automatically provided by the WildFly container. If your resource adapter does not implement the JEE JCA, then connection pooling semantics are left to the user to define on their own.

Dependent Join Pushdown

Dependent joins are a technique used in federation to reduce the cost of cross source joins. Join values from one side of a join are made available to the other side which reduces the number of tuples needed to preform the join. Translators may indicate support for dependent join pushdown via the supportsDependentJoin and supportsFullDependentJoin capabilities. The handling of pushdown dependent join queries can be complicated.

Note	See the JDBC Translator for the reference implementation of dependent join pushdown handling based up the creation temporary tables.

Key Pushdown

The more simplistic mode of dependent join pushdown is to push only the key (equi-join) values to effectively evaluate a semi-join - the full join will still be processed by the engine after the retrieval. The ordering (if present) and all of the non-dependent criteria constructs on the pushdown command **must** be honored. The dependent criteria, which will be a `Comparison` with a `Parameter` (possibly in `Array` form), may be ignored in part or in total to retrieve a superset of the tuples requested.

Pushdown key dependent join queries will be instances of `Select` with the relevant dependent values available via `Select.getDependentValues()`. A dependent value tuple list is associated to Parameters by id via the `Parameter.getDepenentValueId()` identifier. The dependent tuple list provide rows that are referenced by the column positions (available via `Parameter.getValueIndex()`). Care should be taken with the tuple values as they may guaranteed to be ordered, but will be unique with respect to all of the `Parameter` references against the given dependent value tuple list.

Full Pushdown

In some scenarios, typically with small independent data sets or extensive processing above the join that can be pushed to the source, it is advantageous for the source to handle the dependent join pushdown. This feature is marked as supported by the supportsFullDependentJoin capability. Here the source is expected to process the command exactly as specified - the dependent join is not optional

Full pushdown dependent join queries will be instances of `QueryExpression` with the relevant dependent values available via special common table definitions using `QueryExpression.getWith()`. The independent side of a full pushdown join will appear as a common table `WithItem` with a dependent value tuple list available via `WithItem.getDependentValues()`. The dependent value tuples will positionally match the columns defined by `WithItem.getColumns()`. The dependent value tuple list is not guaranteed to be in any particular order.

Executing Commands

Execution Modes

The Teiid query engine uses the "ExecutionFactory" class to obtain the "Execution" interface for the command it is executing. The actual queries themselves are sent to translators in the form of a set of objects, which are further described in Command Language. Refer to Command Language. Translators are allowed to support any subset of the available execution modes.

Execution Interface	Command interface(s)	Description
`ResultSetExecution`	`QueryExpression`	A query corresponding to a SQL SELECT or set query statement.
`UpdateExecution`	`Insert, Update, Delete, BatchedUpdates`	An insert, update, or delete, corresponding to a SQL INSERT, UPDATE, or DELETE command
`ProcedureExecution`	`Call`	A procedure execution that may return a result set and/or output values.

Types of Execution Modes

All of the execution interfaces extend the base `Execution` interface that defines how executions are cancelled and closed. ProcedureExecution also extends ResultSetExecution, since procedures may also return resultsets.

ExecutionContext

The `org.teiid.translator.ExecutionContext` provides a considerable amount of information related to the current execution. An `ExecutionContext` instance is made available to each `Execution` . Specific usage is highlighted in this guide where applicable, but you may use any informational getter method as desired. Example usage would include calling `ExecutionContext.getRequestId()` , `ExecutionContext.getSession()` , etc. for logging purposes.

CommandContext

A `org.teiid.CommandContext` is available via the `ExecutionContext.getCommandContext()` method. The CommandContext contains information about the current user query, including the `VDB` , the ability to add client warnings - `addWarning` , or handle generated keys - `isReturnAutoGeneratedKeys` , `returnGeneratedKeys` , and `getGeneratedKeys` .

Generated Keys

To see if the user query expects generated keys to be returned, consult the `CommandContext.isReturnAutoGeneratedKeys()` method. If you wish to return generated keys, you must first create a `GeneratedKeys` instance to hold the keys with the `returnGeneratedKeys` method passing the column names and types of the key columns. Only one `GeneratedKeys` may be associated with the `CommandContext` at any given time.

Source Hints

The Teiid source meta-hint is used to provide hints directly to source executions via user or transformation queries. See the reference for more on source hints. If specified and applicable, the general and source specific hint will be supplied via the ExecutionContext methods `getGeneralHint` and `getSourceHint`. See the source for the `OracleExecutionFactory` for an example of how this source hint information can be utilized.

ResultSetExecution

Typically most commands executed against translators are QueryExpression. While the command is being executed, the translator provides results via the ResultSetExecution’s "next" method. The "next" method should return null to indicate the end of results. Note: the expected batch size can be obtained from the `ExecutionContext.getBatchSize()` method and used as a hint in fetching results from the EIS.

Update Execution

Each execution returns the update count(s) expected by the update command. If possible BatchedUpdates should be executed atomically. The `ExecutionContext.isTransactional()` method can be used to determine if the execution is already under a transaction.

Procedure Execution

Procedure commands correspond to the execution of a stored procedure or some other functional construct. A procedure takes zero or more input values and can return a result set and zero or more output values. Examples of procedure execution would be a stored procedure in a relational database or a call to a web service.

If a result set is expected when a procedure is executed, all rows from it will be retrieved via the ResultSetExecution interface first. Then, if any output values are expected, they will be retrieved via the getOutputParameterValues() method.

Asynchronous Executions

In some scenarios, a translator needs to execute asynchronously and allow the executing thread to perform other work. To allow asynchronous execution, you should throw a `DataNotAvailableExecption` during a retrieval method, rather than explicitly waiting or sleeping for the results. The `DataNotAvailableException` may take a delay parameter or a `Date` in its constructor to indicate when to poll next for results. Any non-negative delay value indicates the time in milliseconds until the next polling should be performed. The `DataNotAvailableException.NO_POLLING` exception (or any DataNotAvailableException with a negative delay) can be thrown to indicate that the execution will call `ExecutionContext.dataAvailable()` to indicate processing should resume.

Note	A `DataNotAvailableException` should not be thrown by the execute method, as that can result in the execute method being called multiple times.

Note	Since the execution and the associated connection are not closed until the work has completed, care should be taken if using asynchronous executions that hold a lot of state.

A positive retry delay is not a guarantee of when the translator will be polled next. If the `DataNotAvailableException` is consumed while the engine thinks more work can be performed or there are other shorter delays issued from other translators, then the plan may be re-queued earlier than expected. You should simply rethrow a `DataNotAvailableException` if your execution is not yet ready. Alternatively the `DataNotAvailableException` may be marked as strict, which does provide a guarantee that the `Execution` will not be called until the delay has expired or the given `Date` has been reached. Using the `Date` constructor makes the `DataNotAvailableException` automatically strict. Due to engine thread pool contention, platform time resolution, etc. a strict `DataNotAvailableException` is not a real-time guarantee of when the next poll for results will occur, only that it will not occur before then.

Note	If your `ExecutionFactory` returns only asynch executions that perform minimal work, then consider having `ExecutionFactory.isForkable` return false so that the engine knows not to spawn a separate thread for accessing your `Execution`.

Reusable Executions

A translator may return instances of `ReusableExecutions` for the expected Execution objects. There can be one `ReusableExecution` per query executing node in the processing plan. The lifecycle of a `ReusableExecution` is different that a normal `Execution`. After a normal creation/execute/close cycle the `ReusableExecution.reset` is called for the next execution cycle. This may occur indefinitely depending on how many times a processing node executes its query. The behavior of the `close` method is no different than a regular `Execution`, it may not be called until the end of the statement if lobs are detected and any connection associated with the `Execution` will also be closed. When the user command is finished, the `ReusableExecution.dispose()` method will be called.

In general `ReusableExecutions` are most useful for continuous query execution and will also make use of the `ExecutionCotext.dataAvailable()` method for Asynchronous Executions. See the Client Developer's Guide for executing continuous statements. In continuous mode the user query will be continuously re-executed. A `ReusableExecution` allows the same `Execution` object to be associated with the processing plan for a given processing node for the lifetime of the user query. This can simplify asynch resource management, such as establishing queue listeners. Returning a null result from the `next()` method `ReusableExecution` just as with normal `Executions` indicates that the current pushdown command results have ended. Once the `reset()` method has been called, the next set of results should be returned again terminated with a null result.

Bulk Execution

Non batched `Insert, Update, Delete` commands may have multi-valued `Parameter` objects if the capabilities shows support for BulkUpdate. Commands with multi-valued `Parameters` represent multiple executions of the same command with different values. As with BatchedUpdates, bulk operations should be executed atomically if possible.

Command Completion

All normal command executions end with the calling of `close()` on the Execution object. Your implementation of this method should do the appropriate clean-up work for all state created in the Execution object.

Command Cancellation

Commands submitted to Teiid may be aborted in several scenarios:

- Client cancellation via the JDBC API (or other client APIs)
- Administrative cancellation
- Clean-up during session termination
- Clean-up if a query fails during processing Unlike the other execution methods, which are handled in a single-threaded manner, calls to cancel happen asynchronously with respect to the execution thread.

Your connector implementation may choose to do nothing in response to this cancellation message. In this instance, Teiid will call close() on the execution object after current processing has completed. Implementing the cancel() method allows for faster termination of queries being processed and may allow the underlying data source to terminate its operations faster as well.

Extending the ExecutionFactory Class

The main class in the translator implementation is ExecutionFactory. A base class is provided in the Teiid API, so a custom translator must extend `org.teiid.translator.ExecutionFactory` to connect and query an enterprise data source. This extended class must provide a no-arg constructor that can be constructed using Java reflection. This Execution Factory will look similar to the following:

```
package org.teiid.translator.custom;

@Translator(name="custom", description="Connect to My EIS")
public class CustomExecutionFactory extends ExecutionFactory<MyConnectionFactory, MyConnection> {

    public CustomExecutionFactory() {
    }
}
```

Define the annotation `@Translator` on extended "ExecutionFactory" class. This annotation defines the name, which is used as the identifier during deployment, and the description of your translator. This name is what you will be using in the VDB and else where in the configuration to refer to this translator.

ConnectionFactory

Defines the "ConnectionFactory" interface that is defined in resource adapter. This is defined as part of class definition of extended "ExecutionFactory" class. Refer to "MyConnectionFactory" sample in the Developing JEE Connectors chapter.

Connection

Defines the "Connection" interface that is defined in the resource adapter. This is defined as part of class definition of extended "ExecutionFactory" class. Refer to "MyConnection" class sample invthe Developing JEE Connectors chapter.

Configuration Properties

If the translator requires external configuration, that defines ways for the user to alter the behavior of a program, then define an attribute variable in the class and define "get" and "set" methods for that attribute. Also, annotate each "get" method with `@TranslatorProperty` annotation and provide the metadata about the property.

For example, if you need a property called "foo", by providing the annotation on these properties, the Teiid tooling can automatically interrogate and provide a graphical way to configure your Translator while designing your VDB.

```
private String foo = "blah";
@TranslatorProperty(display="Foo property", description="description about Foo")
public String getFoo()
{
   return foo;
}

public void setFoo(String value)
{
   return this.foo = value;
}
```

The `@TranslatorProperty` defines the following metadata that you can define about your property

- display: Display name of the property
- description: Description about the property
- required: The property is a required property
- advanced: This is advanced property; A default value must be provided. A property can not be "advanced" and "required" at same time.
- masked: The tools need to mask the property; Do not show in plain text; used for passwords

Only java primitive (int, boolean), primitive object wrapper (java.lang.Integer), or Enum types are supported as Translator properties. Complex objects are not supported. The default value will be derived from calling the getter method, if available, on a newly constructed instance. All properties *should* have a default value. If there is no applicable default, then the property should be marked in the annotation as `required`. Initialization will fail if a required property value is not provided.

Initializing the Translator

Override and implement the `start` method (be sure to call "super.start()") if your translator needs to do any initializing before it is used by the Teiid engine. This method will be called by Teiid, once after all the configuration properties set above are injected into the class.

Extended Translator Capabilities

These are various methods that typically begin with method signature "supports" on the "ExecutionFactory" class. These methods need to be overridden to describe the execution capabilities of the Translator. Refer to Translator Capabilities for more on these methods.

Execution (and sub-interfaces)

Based on types of executions you are supporting, the following methods need to be overridden to provide implementations for their respective return interfaces.

- `createResultSetExecution` - Override if you are doing read based operation that is returning a rows of results. For ex: select
- `createUpdateExecution` - Override if you are doing write based operations. For ex:insert, update, delete
- `createProcedureExecution` - Overide if you are doing procedure based operations. For ex; stored procedures. This works well for non-relational sources. You can choose to implement all the execution modes or just what you need. See more details on this below.

Metadata

Override and implement the method `getMetadataProcessor()`, if you want to expose the metadata about the source for use in VDBs. This defines the tables, column names, procedures, parameters, etc. for use in the query engine. A sample MetadataProcessor may look like

```
public class MyMetadataProcessor implements MetadataProcessor<Connection> {

     public void process(MetadataFactory mf, Connection conn) {
            Object somedata = connection.getSomeMetadata();
```

```
            Table table = mf.addTable(tableName);
            Column col1 = mf.addColumn("col1", TypeFacility.RUNTIME_NAMES.STRING, table);
            column col2 = mf.addColumn("col2", TypeFacility.RUNTIME_NAMES.STRING, table);

            //add a pushdown function that can also be evaluated in the engine
            Method method = ...
            Function f = mf.addFunction("func", method);

            //add a pushdown aggregate function that can also be evaluated in the engine
            Method aggMethod = ...
            Function af = mf.addFunction("agg", aggMethod);
            af.setAggregateAttributes(new AggregateAttributes());
            ...
       }
}
```

If your MetadataProcessor needs external properties that are needed during the import process, you can define them on MetadataProcessor. For example, to define a import property called "Column Name Pattern", which can be used to filter which columns are defined on the table, can be defined in the code like the following

```
    @TranslatorProperty(display="Column Name Pattern", category=PropertyType.IMPORT, description="Pattern to de
rive column names")
    public String getColumnNamePattern() {
        return columnNamePattern;
    }

    public void setColumnNamePattern(String columnNamePattern) {
        this.columnNamePattern = columnNamePattern;
    }
```

Note the category type. The configuration property defined in the previous section is different from this one. Configuration properties define the runtime behavior of translator, where as "IMPORT" properties define the metadata import behavior, and aid in controlling what metadata is exposed by your translator.

These properties can be automatically injected through "import" properties that can be defined under the <model> construct in the vdb.xml file, like

```
  <vdb name="myvdb" version="1">
     <model name="legacydata" type="PHYSICAL">
        <property name="importer.ColumnNamePattern" value="col*"/>
        ....
        <source name = .../>
     </model>
  </vdb>
```

Extension Metadata Properties

There may be times when implementing a custom translator, the built in metadata about your schema is not enough to process the incoming query due to variance of semantics with your source query. To aid this issue, Teiid provides a mechanism called "Extension Metadata", which is a mechanism to define custom properties and then add those properties on metadata object (table, procedure, function, column, index etc.). For example, in my custom translator a table represents a file on disk. I could define a extension metadata property as

```
public class MyMetadataProcessor implements MetadataProcessor<Connection> {
     public static final String NAMESPACE = "{http://my.company.corp}";

      @ExtensionMetadataProperty(applicable={Table.class}, datatype=String.class, display="File name", descript
ion="File Name", required=true)
     public static final String FILE_PROP = NAMESAPCE+"FILE";
```

```
    public void process(MetadataFactory mf, Connection conn) {
        Object somedata = connection.getSomeMetadata();

        Table table = mf.addTable(tableName);
        table.setProperty(FILE_PROP, somedata.getFileName());

        Column col1 = mf.addColumn("col1", TypeFacility.RUNTIME_NAMES.STRING, table);
        column col2 = mf.addColumn("col2", TypeFacility.RUNTIME_NAMES.STRING, table);

    }
}
```

The `@ExtensionMetadataProperty` defines the following metadata that you can define about your property

- applicable: Metadata object this is applicable on. This is array of metadata classes like Table.class, Column.class.
- datatype: The java class indicating the data type
- display: Display name of the property
- description: Description about the property
- required: Indicates if the property is a required property

How this is used?

When you define an extension metadata property like above, during the runtime you can obtain the value of that property. If you get the query object which contains `SELECT * FROM MyTable', MyTable will be represented by an object called "NamedTable". So you can do the following

```
for (TableReference tr:query.getFrom()) {
    NamedTable t = (NameTable) tr;
    Table table = t.getMetadataObject();
    String file = table.getProperty(FILE_PROP);
    ..
}
```

Now you have accessed the file name you set during the construction of the Table schema object, and you can use this value however you seem feasible to execute your query. With the combination of built in metadata properties and extension metadata properties you can design and execute queries for a variety of sources.

Logging

Teiid provides `org.teiid.logging.LogManager` class for logging purposes. Create a logging context and use the LogManager to log your messages. These will be automatically sent to the main Teiid logs. You can edit the "jboss-log4j.xml" inside "conf" directory of the WildFly’s profile to add the custom context. Teiid uses Log4J as its underlying logging system.

Exceptions

If you need to bubble up any exception use `org.teiid.translator.TranslatorException` class.

Large Objects

This section examines how to use facilities provided by the Teiid API to use large objects such as blobs, clobs, and xml in your Translator.

Data Types

Teiid supports three large object runtime data types: blob, clob, and xml. A blob is a "binary large object", a clob is a "character larg object", and "xml" is a "xml document". Columns modeled as a blob, clob, or xml are treated similarly by the translator framework to support memory-safe streaming.

Why Use Large Object Support?

Teiid allows a Translator to return a large object through the Teiid translator API by just returning a reference to the actual large object. Access to that LOB will be streamed as appropriate rather than retrieved all at once. This is useful for several reasons:

1. Reduces memory usage when returning the result set to the user.
2. Improves performance by passing less data in the result set.
3. Allows access to large objects when needed rather than assuming that users will always use the large object data.
4. Allows the passing of arbitrarily large data values. However, these benefits can only truly be gained if the Translator itself does not materialize an entire large object all at once. For example, the Java JDBC API supports a streaming interface for blob and clob data.

Handling Large Objects

The Translator API automatically handles large objects (Blob/Clob/SQLXML/Geometry/JSON) through the creation of special purpose wrapper objects when it retrieves results.

Once the wrapped object is returned, the streaming of LOB is automatically supported. These LOB objects then can for example appear in client results, in user defined functions, or sent to other translators.

A Execution is usually closed and the underlying connection is either closed/released as soon as all rows for that execution have been retrieved. However, LOB objects may need to be read after their initial retrieval of results. When LOBs are detected the default closing behavior is prevented by setting a flag via the `ExecutionContext.keepAlive` method.

When the "keepAlive" alive flag is set, then the execution object is only closed when user’s Statement is closed.

```
executionContext.keepExecutionAlive(true);
```

Inserting or Updating Large Objects

LOBs will be passed to the Translator in the language objects as Literal containing a java.sql.Blob, java.sql.Clob, or java.sql.SQLXML. You can use these interfaces to retrieve the data in the large object and use it for insert or update.

Translator Capabilities

The `ExecutionFactory` class defines all the methods that describe the capabilities of a Translator. These are used by the Connector Manager to determine what kinds of commands the translator is capable of executing. A base `ExecutionFactory` class implements all the basic capabilities methods, which says your translator does not support any capabilities. Your extended `ExecutionFactory` class must override the the necessary methods to specify which capabilities your translator supports. You should consult the debug log of query planning (set showplan debug) to see if desired pushdown requires additional capabilities.

Capability Scope

Note capabilities are determined and cached for the lifetime of the translator. Capabilities based on connection/user are not supported.

Capabilities

The following table lists the capabilities that can be specified in the `ExecutionFactory` class.

Table 1. **Available Capabilities**

Capability	Requires	Description
SelectDistinct		Translator can support SELECT DISTINCT in queries.
SelectExpression		Translator can support SELECT of more than just column references.
SelectExpressionArrayType	SelectExpression, ArrayType	Translator can support SELECT of array expressions.
SelectWithoutFrom		Translator can support a SELECT of scalar values without a FROM clause
AliasedTable		Translator can support Tables in the FROM clause that have an alias.
InnerJoins		Translator can support inner and cross joins
SelfJoins	AliasedGroups and at least one of the join type supports.	Translator can support a self join between two aliased versions of the same Table.
OuterJoins		Translator can support LEFT and RIGHT OUTER JOIN.
FullOuterJoins		Translator can support FULL OUTER JOIN.
DependentJoins	Base join and criteria support	Translator supports key set dependent join pushdown. See Dependent Join Pushdown. When set the MaxDependentInPredicates and MaxInCriteriaSize values are not used by the engine, rather all independent values are made available to the pushdown command.

FullDependentJoins	Base join and criteria support	Translator supports full dependent join pushdown. See Dependent Join Pushdown. When set the MaxDependentInPredicates and MaxInCriteriaSize values are not used by the engine, rather the entire independent dataset is made available to the pushdown command.
LateralJoin		Translator supports lateral join pushdown with sideways correlation.
LateralJoinCondition	LateralJoin	Translator supports lateral join pushdown with a join condition.
OnlyLateralJoinProcedure	LateralJoin	Translator supports only lateral join to a procedure or table valued function.
SubqueryInOn	Join and base subquery support, such as ExistsCriteria	Translator can support subqueries in the ON clause. Defaults to true.
InlineViews	AliasedTable	Translator can support a named subquery in the FROM clause.
ProcedureTable		Translator can support a table that returns a table in the FROM clause.
ProcedureParameterExpression		Translator can support an expression, not just a literal, as a procedure parameter.
BetweenCriteria		Not currently used - between criteria is rewriten as compound comparisions.
CompareCriteriaEquals		Translator can support comparison criteria with the operator `=` .
CompareCriteriaOrdered		Translator can support comparison criteria with the operator `⇒` or `⇐` .
CompareCriteriaOrderedExclusive		Translator can support comparison criteria with the operator `>` or `<` . Defaults to CompareCriteriaOrdered
LikeCriteria		Translator can support LIKE criteria.
LikeCriteriaEscapeCharacter	LikeCriteria	Translator can support LIKE criteria with an ESCAPE character clause.
SimilarTo		Translator can support SIMILAR TO criteria.
LikeRegexCriteria		Translator can support LIKE_REGEX criteria.
InCriteria	MaxInCriteria	Translator can support IN predicate criteria.
InCriteriaSubquery		Translator can support IN predicate criteria where values are supplied by a subquery.

IsNullCriteria		Translator can support IS NULL predicate criteria.
OrCriteria		Translator can support the OR logical criteria.
NotCriteria		Translator can support the NOT logical criteria. IMPORTANT: This capability also applies to negation of predicates, such as specifying IS NOT NULL, `<` (not `⇒`), `>` (not `⇐`), etc.
ExistsCriteria		Translator can support EXISTS predicate criteria.
QuantifiedCompareCriteriaAll		Translator can support a quantified comparison criteria using the ALL quantifier.
QuantifiedCompareCriteriaSome		Translator can support a quantified comparison criteria using the SOME or ANY quantifier.
OnlyLiteralComparison		If only Literal comparisons (equality, ordered, like, etc.) are supported for non-join conditions.
Convert(int fromType, int toType)		Used for fine grained control of convert/cast pushdown. The `ExecutionFactory.getSupportedFunctions()` should contain `SourceSystemFunctions.CONVERT` . This method can then return false to indicate a lack of specific support. See `TypeFacility.RUNTIME_CODES` for the possible type codes. The engine will does not care about an unnecessary conversion where fromType == toType. By default lob conversion is disabled.
OrderBy		Translator can support the ORDER BY clause in queries.
OrderByUnrelated	OrderBy	Translator can support ORDER BY items that are not directly specified in the select clause.
OrderByNullOrdering	OrderBy	Translator can support ORDER BY items with NULLS FIRST/LAST.
OrderByWithExtendedGrouping	OrderBy	Translator can support ORDER BY directly over a GROUP BY with an extended grouping element such as a ROLLUP.
GroupBy		Translator can support an explicit GROUP BY clause.
GroupByRollup	GroupBy	Translator can support GROUP BY (currently a single) ROLLUP.
GroupByMultipleDistinctAggregates	GroupBy	Translator can support GROUP BY to create multiple distinct aggregates (See IMPALA-110).

Having	GroupBy	Translator can support the HAVING clause.
AggregatesAvg		Translator can support the AVG aggregate function.
AggregatesCount		Translator can support the COUNT aggregate function.
AggregatesCountBig	AggregatesCount, AggregatesCountStar	Translator supports a separate COUNT function that returns a long value. If false COUNT will be pushed instead.
AggregatesCountStar		Translator can support the COUNT(*) aggregate function.
AggregatesDistinct	At least one of the aggregate functions.	Translator can support the keyword DISTINCT inside an aggregate function. This keyword indicates that duplicate values within a group of rows will be ignored.
AggregatesMax		Translator can support the MAX aggregate function.
AggregatesMin		Translator can support the MIN aggregate function.
AggregatesSum		Translator can support the SUM aggregate function.
AggregatesEnhancedNumeric		Translator can support the VAR_SAMP, VAR_POP, STDDEV_SAMP, STDDEV_POP aggregate functions.
StringAgg		Translator can support the string_agg aggregate function.
ListAgg		Translator can support a restricted form (matching Oracle’s listagg) of the string_agg aggregate function.
ScalarSubqueries		Translator can support the use of a subquery in a scalar context (wherever an expression is valid).
ScalarSubqueryProjection	ScalarSubqueries	Translator can support the use of a projected scalar subquery.
CorrelatedSubqueries	At least one of the subquery pushdown capabilities.	Translator can support a correlated subquery that refers to an elementin the outer query.
CorrelatedSubqueryLimit	CorrelatedSubqueries	Defaults to CorrelatedSubqueries support. Translator can support a correlated subquery with a limit clause.
CaseExpressions		Not currently used - simple case is rewriten as searched case.

SearchedCaseExpressions		Translator can support `searched` CASE expressions anywhere that expressions are accepted.
Unions		Translator support UNION and UNION ALL
Intersect		Translator supports INTERSECT
Except		Translator supports Except
SetQueryOrderBy	Unions, Intersect, or Except	Translator supports set queries with an ORDER BY
SetQueryLimitOffset	(Unions, Intersect, or Except) and (RowLimit or RowOffset)	Translator supports set queries with a LIMIT and/or OFFSET which is determined by the respective RowLimit and RowOffset capability. Defaults to true if RowLimit or RowOffset is supported.
RowLimit		Translator can support the limit portion of the limit clause
RowOffset		Translator can support the offset portion of the limit clause
FunctionsInGroupBy	GroupBy	Translator can support non-column reference grouping expressions.
InsertWithQueryExpression		Translator supports INSERT statements with values specified by an QueryExpression.
BatchedUpdates		Translator supports a batch of INSERT, UPDATE and DELETE commands to be executed together.
BulkUpdate		Translator supports updates with multiple value sets
CommonTableExpressions		Translator supports the WITH clause.
SubqueryCommonTableExpressions	CommonTableExpressions	Translator supports a WITH clause in subqueries.
RecursiveCommonTableExpressions	CommonTableExpressions	Translator supports recursive common table expressions
ElementaryOlapOperations		Translator supports window functions and analytic functions RANK, DENSE_RANK, and ROW_NUMBER.
WindowFrameClause	ElementaryOlapOperations	Translator supports window frame RANGE/ROWS clause. Defaults to ElementaryOlapOperations support value.
WindowOrderByWithAggregates	ElementaryOlapOperations	Translator supports windowed aggregates with a window order by clause.

WindowDistinctAggregates	ElementaryOlapOperations, AggregatesDistinct	Translator supports windowed distinct aggregates.
AdvancedOlapOperations	ElementaryOlapOperations	Translator supports aggregate conditions.
WindowFunctionCumeDist		Translator supports CUME_DIST window function. Defaults to the support value for ElementaryOlapOperations
WindowFunctionPercentDist		Translator supports PERCENT_DIST window function. Defaults to the support value for ElementaryOlapOperations
WindowFunctionNtile		Translator supports NTILE window function. Defaults to the support value for ElementaryOlapOperations
WindowFunctionNthValue		Translator supports NTH_VALUE window function. Defaults to the support value for ElementaryOlapOperations
OnlyFormatLiterals	function support for a parse/format function and an implementation of the supportsFormatLiteral method.	Translator supports only literal format patterns that must be validated by the supportsFormatLiteral method.
FormatLiteral(String literal, Format type)	OnlyFormatLiterals	Translator supports the given literal format string.
ArrayType		Translator supports the push down of array values.
OnlyCorrelatedSubqueries	CorrelatedSubqueries	Translator ONLY supports correlated subqueries. Uncorrelated scalar and exists subqueries will be pre-evaluated prior to push-down.
SelectWithoutFrom	SelectExpressions	Translator supports selecting values without a FROM clause, e.g. SELECT 1.
Upsert		Translator supports an upsert style insert.
OnlyTimestampAddLiteral	function support for a timestampadd function.	Translator supports only a literal interval value.
MultipleOpenExecutions		Translator supports multiple open executions against a single connection. If false, in transactional scenarios the execution will be thread bound.
GeographyType		Translator supports the geograpy type variations of ST_ geospatial functions.

Note that any pushdown subquery must itself be compliant with the Translator capabilities.

Command Form

The method `ExecutionFactory.useAnsiJoin()` should return true if the Translator prefers the use of ANSI style join structure for join trees that contain only INNER and CROSS joins.

The method `ExecutionFactory.requiresCriteria()` should return true if the Translator requires criteria for any Query, Update, or Delete. This is a replacement for the model support property `Where All`.

Scalar Functions

The method `ExecutionFactory.getSupportedFunctions()` can be used to specify which system/user defined scalar and user defined aggregate functions the Translator supports. The constants interface `org.teiid.translator.SourceSystemFunctions` contains the string names of all possible built-in pushdown functions, which includes the four standard math operators: +, -, *, and /.

Not all system functions appear in SourceSystemFunctions, since some system functions will always be evaluated in Teiid, are simple aliases to other functions, or are rewritten to a more standard expression.

This documentation for system functions can be found at Scalar Functions. If the Translator states that it supports a function, it must support all type combinations and overloaded forms of that function.

A translator may also indicate support for scalar functions that are intended for pushdown evaluation by that translator, but are not registered as user defined functions via a model/schema. These pushdown functions are reported to the engine via the `ExecutionFactory.getPushDownFunctions()` list as `FunctionMethod` metadata objects. The `FuncitonMethod` representation allow the translator to control all of the metadata related to the function, including type signature, determinism, varargs, etc. The simplest way to add a pushdown function is with a call to `ExecutionFactory.addPushDownFunction` :

```
FunctionMethod addPushDownFunction(String qualifier, String name, String returnType, String...paramTypes)
```

This resulting function will be known as sys.qualifier.name, but can be called with just name as long as the function name is unique. The returned `FunctionMethod` object may be further manipulated depending upon the needs of the source. An example of adding a custom concat vararg function in an `ExecutionFactory` subclass:

```
public void start() throws TranslatorException {
  super.start();
  FunctionMethod func = addPushDownFunction("oracle", "concat", "string", "string", "string");
  func.setVarArgs(true);
  ...
}
```

Physical Limits

The method `ExecutionFactory.getMaxInCriteriaSize()` can be used to specify the maximum number of values that can be passed in an IN criteria. This is an important constraint as an IN criteria is frequently used to pass criteria between one source and another using a dependent join.

The method `ExecutionFactory.getMaxDependentInPredicates()` is used to specify the maximum number of IN predicates (of at most MaxInCriteriaSize) that can be passed as part of a dependent join. For example if there are 10000 values to pass as part of the dependent join and a MaxInCriteriaSize of 1000 and a MaxDependentInPredicates setting of 5, then the dependent join logic will form two source queries each with 5 IN predicates of 1000 values each combined by OR.

The method `ExecutionFactory.getMaxFromGroups()` can be used to specify the maximum number of FROM Clause groups that can used in a join. -1 indicates there is no limit.

The method `ExecutionFactory.getMaxProjectedColumns()` can be used to specify the maximum number of columns or expressions in the select clause. -1 indicates there is no limit.

Update Execution Modes

The method `ExecutionFactory.supportsBatchedUpdates()` can be used to indicate that the Translator supports executing the `BatchedUpdates` command.

The method `ExecutionFactory.supportsBulkUpdate()` can be used to indicate that the Translator accepts update commands containg multi valued Literals.

Note that if the translator does not support either of these update modes, the query engine will compensate by issuing the updates individually.

Default Behavior

The method `ExecutionFactory.getDefaultNullOrder()` specifies the default null order. Can be one of UNKNOWN, LOW, HIGH, FIRST, LAST. This is only used if ORDER BY is supported, but null ordering is not.

The method `ExecutionFactory.getCollation()` specifies the default collation. If set to a value that does not match the collation locale defined by org.teiid.collationLocale, then some ordering may not be pushed down.

The method `ExecutionFactory.getRequiredLikeEscape()` specifies the required like escape character. Used only when a source supports a specific escape.

Use of Connections

Method	Description	Default
is/setSourceRequired	True indicates a source connection is required for fetching the metadata of the source or executing queries.	true
is/setSourceRequiredForMetadata	True indicates a source connection is required for fetching the metadata of the source.	SourceRequired

Transaction Behavior

ExecutionFactory.get/setTransactionSupport specifies the highest level of transaction supported by connections to the source. This is used as a hint to the engine for deciding when to start a transaction in the autoCommitTxn=DETECT mode. Defaults to XA.

Translator Properties

During translator development, a translator developer can define three (3) different types of property sets that can help customize the behavior of the translator. The sections below describes each one.

Translator Override Properties

On the "ExecutionFactory" class a translator developer can define any number of "getter/setter" methods with the @TranslatorProperty annotation. These properties (also referred to a execution properties) can be used for extending the capabilities of the translator. It is important to define default values for all these properties, as these properties are being defined to change the default behavior of the translator. If needed, the values for these properties are supplied in the vdb during the deploy time when the translator is used to represent vdb’s model. A sample example is given below:

```
@TranslatorProperty(display="Copy LOBs",description="If true, returned LOBs will be copied, rather than streame
d from the source",advanced=true)
public boolean isCopyLobs() {
    return copyLobs;
}

public void setCopyLobs(boolean copyLobs) {
    this.copyLobs = copyLobs;
```

at runtime these properties can be defined in the vdb as

```
CREATE DATABASE vdb;
USE DATABASE vdb;
CREATE FOREIGN DATA WRAPPER "my-translator-override" TYPE "my-translator" OPTIONS (CopyLobs 'true');
CREATE SERVER connector FOREIGN DATA WRAPPER "my-translator-override";
CREATE SCHEMA PM1 SERVER connector;
SET SCHEMA PM1;
IMPORT FROM SERVER connector INTO PM1;
```

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<vdb name="vdb" version="1">
    <model name="PM1">
        <source name="connector" translator-name="my-translator-override" />
    </model>
    <translator name="my-translator-override" type="my-translator">
        <property name="CopyLobs" value="true" />
    </translator>
</vdb>
```

Metadata Import Properties

If a translator is defining schema information based on the physical source (i.e. implementing *getMetadata* method on *ExecutionFactory*) it is connected to, then import properties provide a way to customize the behavior of the import process. For example, in the JDBC translator users can exclude certain tables that match a regular expression etc. To define a import property, the @TranslatorPropery annotation is used on any getter/setter method on the "ExecutionFactory" class or any class that implements the "MetadataProcessor" interface, with *category* property defined as "PropertyType.IMPORT". For example.

```
@Translator(name = "my-translator", description = "My Translator")
public class MyExecutionFactory extends ExecutionFactory<ConnectionFactory, MyConnection> {
...
```

```
    public MetadataProcessor<C> getMetadataProcessor() {
        return MyMetadataProcessor();
    }
}

public MyMetadataProcessor implements MetadataProcessor<MyConnection> {

    public void process(MetadataFactory metadataFactory, MyConnection connection) throws TranslatorException{
        // schema generation code here
    }

    @TranslatorProperty(display="Header Row Number", category=PropertyType.IMPORT, description="Row number that
 contains the header information")
    public int getHeaderRowNumber() {
        return headerRowNumber;
    }

    public void setHeaderRowNumber(int headerRowNumber) {
        this.headerRowNumber = headerRowNumber;
    }
}
```

Below is an example showing how to use import properties with a vdb file

```
CREATE DATABASE vdb;
USE DATABASE vdb;
CREATE SERVER connector FOREIGN DATA WRAPPER "my-translator";
CREATE SCHEMA PM1 SERVER connector OPTIONS ("importer.HeaderRowNumber" '12');
SET SCHEMA PM1;
IMPORT FROM SERVER connector INTO PM1 OPTIONS ("importer.HeaderRowNumber" '12');
```

Note that the import properties in DDL may be on either the SERVER or the IMPORT statement.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<vdb name="vdb" version="1">
    <model name="PM1">
        <property name="importer.HeaderRowNumber" value="12"/>
        <source name="connector" translator-name="my-translator" />
    </model>
</vdb>
```

Extension Metadata Properties

During the execution of the command in translator, a translator is responsible to convert Teiid supplied SQL command into data source specific query. Most of times this conversion is not a trivial task can be converted from one form to another. There are many cases built-in metadata is not sufficient and additional metadata about source is useful to form a request to the underlying physical source system. Extension Metadata Properties one such mechanism to fill the gap in the metadata. These can be defined specific for a given translator.

A translator is a plugin, that is communicating with Teiid engine about it’s source with it’s metadata. Metadata in this context is definitions of Tables, Columns, Procedures, Keys etc. This metadata can be decorated with additional custom metadata and fed to Teiid query engine. Teiid query engine keeps this extended metadata intact along with its schema objects, and when a user query is submitted to the the translator for execution, this extended metadata can be retrieved for making decisions in the translator code.

Extended properties are defined using annotation class called @ExtensionMetadataProperty on the fields in your "MetadataProcessor" or "ExcutionFactory" classes.

For example, say translator requires a "encoding" property on Table, to do the correct un-marshaling of data, this property can be defined as

```
public class MyMetadataProcessor implements MetadataProcessor<MyConnection> {
    public static final String URI = "{http://www.teiid.org/translator/mytranslator/2014}";

    @ExtensionMetadataProperty(applicable=Table.class, datatype=String.class, display="Encoding", description="
Encoding", required=true)
    public static final String ENCODING = URI+"encode";

    public void process(MetadataFactory mf, FileConnection conn) throws TranslatorException {
        ..
        Table t = mf.addTable(tableName);
        t.setProperty(ENCODING, "UTF-16");

        // add columns etc.
        ..
    }
}
```

Now during the execution, on the COMMAND object supplied to the "Execution" class, user can

```
    Select select = (Select)command;
    NamedTable tableReferece = select.getFrom().get(0);
    Table t = tableReference.getMetadataObject();
    String encoding = t.getProperty(MyMetadataProcessor.ENCODING, false);

    // use the encoding value as needed to marshal or unmarshal data
```

Extending The JDBC Translator

The JDBC Translator can be extended to handle new JDBC drivers and database versions. This is one of the most common needs of custom Translator development. This chapter outlines the process by which a user can modify the behavior of the JDBC Translator for a new source, rather than starting from scratch.

To design a JDBC Translator for any RDMS that is not already provided by the Teiid, extend the `org.teiid.translator.jdbc.JDBCExecutionFactory` class in the "translator-jdbc" module. There are three types of methods that you can override from the base class to define the behavior of the Translator.

Extension	Purpose
Capabilities	Specify the SQL syntax and functions the source supports.
SQL Translation	Customize what SQL syntax is used, how source-specific functions are supported, how procedures are executed.
Results Translation	Customize how results are retrieved from JDBC and translated.

Table of Contents

Capabilities Extension

This extension must override the methods that begin with "supports" that describe translator capabilities. Refer to Command Language#Translator Capabilities for all the available translator capabilities.

The most common example is adding support for a scalar function – this requires both declaring that the translator has the capability to execute the function and often modifying the SQL Translator to translate the function appropriately for the source.

Another common example is turning off unsupported SQL capabilities (such as outer joins or subqueries) for less sophisticated JDBC sources.

SQL Translation Extension

The JDBCExcecutionFactory provides several methods to modify the command and the string form of the resulting syntax before it is sent to the JDBC driver, including:

- Change basic SQL syntax options. See the useXXX methods, e.g. useSelectLimit returns true for SQLServer to indicate that limits are applied in the SELECT clause.
- Register one or more FunctionModifiers that define how a scalar function should be modified or transformed.
- Modify a LanguageObject. - see the translate, translateXXX, and FunctionModifiers.translate methods. Modify the passed in object and return null to indicate that the standard syntax output should be used.

- Change the way SQL strings are formed for a LanguageObject. - - see the translate, translateXXX, and FunctionModifiers.translate methods. Return a list of parts, which can contain strings and LanguageObjects, that will be appended in order to the SQL string. If the in coming LanguageObject appears in the returned list it will not be translated again. Refer to Using FunctionModifiers.

Results Translation Extension

The JDBCExecutionFactory provides several methods to modify the java.sql.Statement and java.sql.ResultSet interactions, including:

1. Overriding the createXXXExecution to subclass the corresponding JDBCXXXExecution. The JDBCBaseExecution has protected methods to get the appropriate statement (getStatement, getPreparedStatement, getCallableStatement) and to bind prepared statement values bindPreparedStatementValues.
2. Retrieve values from the JDBC ResultSet or CallableStatement - see the retrieveValue methods.

Adding Function Support

Refer to User Defined Functions for adding new functions to Teiid. This example will show you how to declare support for the function and modify how the function is passed to the data source.

Following is a summary of all coding steps in supporting a new scalar function:

1. Override the capabilities method to declare support for the function (REQUIRED)
2. Implement a FunctionModifier to change how a function is translated and register it for use (OPTIONAL) There is a capabilities method getSupportedFunctions() that declares all supported scalar functions.

An example of an extended capabilities class to add support for the "abs" absolute value function:

```
package my.connector;

import java.util.ArrayList;
import java.util.List;

public class ExtendedJDBCExecutionFactory extends JDBCExecutionFactory
{
   @Override
   public List getSupportedFunctions()
   {
      List supportedFunctions = new ArrayList();
      supportedFunctions.addAll(super.getSupportedFunctions());
      supportedFunctions.add("ABS");
      return supportedFunctions;
   }
}
```

In general, it is a good idea to call super.getSupportedFunctions() to ensure that you retain any function support provided by the translator you are extending.

This may be all that is needed to support a Teiid function if the JDBC data source supports the same syntax as Teiid. The built-in SQL translation will translate most functions as: "function(arg1, arg2,…)".

Using FunctionModifiers

In some cases you may need to translate the function differently or even insert additional function calls above or below the function being translated. The JDBC translator provides an abstract class `FunctionModifier` for this purpose.

During the start method a modifier instance can be registered against a given function name via a call to `JDBCExecutionFactory.registerFunctionModifier`.

The FunctionModifier has a method called `translate`. Use the translate method to change the way the function is represented.

An example of overriding the translate method to change the MOD(a, b) function into an infix operator for Sybase (a % b). The translate method returns a list of strings and language objects that will be assembled by the translator into a final string. The strings will be used as is and the language objects will be further processed by the translator.

```
public class ModFunctionModifier extends FunctionModifier
{
   public List translate(Function function)
   {
      List parts = new ArrayList();
      parts.add("(");
      Expression[] args = function.getParameters();
      parts.add(args[0]);
      parts.add(" % ");
      parts.add(args[1]);
      parts.add(")");
      return parts;
   }
}
```

In addition to building your own FunctionModifiers, there are a number of pre-built generic function modifiers that are provided with the translator.

Modifier	Description
AliasModifier	Handles simply renaming a function ("ucase" to "upper" for example)
EscapeSyntaxModifier	Wraps a function in the standard JDBC escape syntax for functions: {fn xxxx()}

To register the function modifiers for your supported functions, you must call the `ExecutionFactory.registerFunctionModifier(String name, FunctionModifier modifier)` method.

```
public class ExtendedJDBCExecutionFactory extends JDBCExecutionFactory
{
   @Override
   public void start()
   {
      super.start();

      // register functions.
      registerFunctionModifier("abs", new MyAbsModifier());
      registerFunctionModifier("concat", new AliasModifier("concat2"));
   }
}
```

Support for the two functions being registered ("abs" and "concat") must be declared in the capabilities as well. Functions that do not have modifiers registered will be translated as usual.

Installing Extensions

Once you have developed an extension to the JDBC translator, you must install it into the Teiid Server. The process of packaging or deploying the extended JDBC translators is exactly as any other other translator. Since the RDMS is accessible already through its JDBC driver, there is no need to develop a resource adapter for this source as WildFly provides a wrapper JCA connector (DataSource) for any JDBC driver.

Refer to Packaging and Deployment for more details.

Delegating Translator

In some instances you may wish to extend several different kinds of translators with the same functionality. Rather than create separate subclasses for each extension, you can use the delegating translator framework which provides you with a proxying mechanism to override translator behavior. It implement a delegating translator, your common translator logic should be added to a subclass of BaseDelegatingExecutionFactory where you can override any of the delegation methods to perform whatever logic you want.

Example BaseDelegatingExecutionFactory Subclass

```
@Translator(name="custom-delegator")
public class MyTranslator extends BaseDelegatingExecutionFactory<Object, Object> {

    @Override
    public Execution createExecution(Command command,
            ExecutionContext executionContext, RuntimeMetadata metadata,
            Object connection) throws TranslatorException {
        if (command instanceof Select) {
            //modify the command or return a different execution
            ...

        }
        //the super call will be to the delegate instance
        return super.createExecution(command, executionContext, metadata, connection);
    }
    ...
}
```

You will bundle and deploy your custom delegating translator is just like any other custom translator development. To you use your delegating translator in a vdb, you define a translator override that wires in the delegate.

Example Translator Override

```
<translator type="custom-delegator" name="my-translator">

     <property value="delegateName" name="name of the delegate instance"/>

     <!-- any custom properties you may have on your custom translator -->

</translator>
```

From the previous example the translator type is custom-delegator. Now my-translator can be used as a translator-name on a source and will proxy all calls to whatever delegate instance you assign.

Note	Note that the delegate instance can be any translator instance, whether configured by it’s own translator entry or just the name of a standard translator type.

Packaging

Once the "ExecutionFactory" class is implemented, package it in a JAR file. Then add the following named file in "META-INF/services/org.teiid.translator.ExecutionFactory" with contents specifying the name of your main Translator file. Note that, the name must exactly match to above. This is java’s standard service loader pattern. This will register the Translator for deployment when the jar is deployed into WildFly.

```
org.teiid.translator.custom.CustomExecutionFactory
```

Adding Dependent Modules

Add a MANIFEST.MF file in the META-INF directory, and the core Teiid API dependencies for resource adapter with the following line.

```
Dependencies: org.jboss.teiid.common-core,org.jboss.teiid.api,javax.api
```

If your translator depends upon any other third party jar files, ensure a module exists and add the module name to the above MANIFEST.MF file.

Deployment

A translator JAR file can be deployed into Teiid Server in two different ways

As WildFly module

Create a module under "jboss-as/modules" directory and define the translator name and module name in the teiid subsystem in *standalone-teiid.xml* file or *domain-teiid.xml* file and restart the server. The dependent Teiid or any other java class libraries must be defined in module.xml file of the module. For production profiles this is recommended.

As JAR deployment

For development time or quick deployment you can deploy the translator JAR using the CLI or AdminAPI or admin console programs. When you deploy in JAR form the dependencies to Teiid java libraries and any other third party libraries must be defined under *META-INF/MANIFEST.MF* file.

User Defined Functions

If you need to extend Teiid's scalar or aggregate function library, then Teiid provides a means to define custom or User Defined Functions(UDF).

The following are used to define a UDF.

- *Function Name* When you create the function name, keep these requirements in mind:
  - You cannot overload existing Teiid System functions.
  - The function name must be unique among user-defined functions in its model for the number of arguments. You can use the same function name for different numbers of types of arguments. Hence, you can overload your user-defined functions.
  - The function name cannot contain the `.' character.
  - The function name cannot exceed 255 characters.
- *Input Parameters*- defines a type specific signature list. All arguments are considered required.
- *Return Type*- the expected type of the returned scalar value.
- *Pushdown*- can be one of REQUIRED, NEVER, ALLOWED. Indicates the expected pushdown behavior. If NEVER or ALLOWED are specified then a Java implementation of the function should be supplied. If REQUIRED is used, then user must extend the Translator for the source and add this function to its pushdown function library.
- *invocationClass/invocationMethod*- optional properties indicating the method to invoke when the UDF is not pushed down.
- *Deterministic*- if the method will always return the same result for the same input parameters. Defaults to false. It is important to mark the function as deterministic if it returns the same value for the same inputs as this will lead to better performance. See also the Relational extension boolean metadata property "deterministic" and the DDL OPTION property "determinism". Defaults to false. It is important to mark the function as deterministic if it returns the same value for the same inputs as this will lead to better performance. See also the Relational extension boolean metadata property "deterministic" and the DDL OPTION property "determinism".

Even *Pushdown* required functions need to be added as a UDF to allow Teiid to properly parse and resolve the function. *Pushdown* scalar functions differ from normal user-defined functions in that no code is provided for evaluation in the engine. An exception will be raised if a pushdown required function cannot be evaluated by the appropriate source.

Source Supported Functions

While Teiid provides an extensive scalar function library, it contains only those functions that can be evaluated within the query engine. In many circumstances, especially for performance, a source function allows for calling a source specific function. The semantics of defining the source function as similar or same to one of defining the UDF.

For example, suppose you want to use the Oracle-specific functions *score* and *contains* like:

```
SELECT score(1), ID, FREEDATA FROM Docs WHERE contains(freedata, 'nick', 1) > 0
```

The *score* and *contains* functions are not part of built-in scalar function library. While you could write your own custom scalar function to mimic their behavior, it's more likely that you would want to use the actual Oracle functions that are provided by Oracle when using the Oracle Free Text functionality.

In order to configure Teiid to *push* the above function evaluation to Oracle, Teiid provides a few different ways one can configure their instance.

Extending the Translator

The ExecutionFactory.getPushdownFunctions method can be used to describe functions that are valid against all instances of a given translator type. The function names are expected to be prefixed by the translator type, or some other logical grouping, e.g. salesforce.includes. The full name of the function once imported into the system will qualified by the SYS schema, e.g. SYS.salesforce.includes.

Any functions added via these mechanisms do not need to be declared in ExecutionFactory.getSupportedFunctions. Any of the additional handling, such as adding a FunctionModifier, covered above is also applicable here. All pushdown functions will have function name set to only the simple name. Schema or other qualification will be removed. Handling, such as function modifiers, can check the function metadata if there is the potential for an ambiguity.

For example, to extend the Oracle Connector

- *Required*- extend the OracleExecutionFactory and add *SCORE* and *CONTAINS* as supported pushdown functions by either overriding or adding additional functions in "getPushDownFunctions" method. For this example, we'll call the class `MyOracleExecutionFactory` . Add the `org.teiid.translator.Translator` annotation to the class, e.g. `@Translator(name="myoracle")`
- Optionally register new FunctionModifiers on the start of the ExecutionFactory to handle translation of these functions. Given that the syntax of these functions is same as other typical functions, this probably isn't needed - the default translation should work.
- Create a new translator JAR containing your custom ExecutionFactory. Refer to Packaging and Deployment for instructions on using the JAR file. Once this is extended translator is deployed in the Teiid Server, use "myoracle" as translator name instead of the "oracle" in your VDB's Oracle source configuration.

If you source handing of the function can be described by simple parameter substitution into a string, then you may not need to extend the translator for a source specific function. You can use the extension property teiid_rel:native-query to define the syntax handling - see also DDL Metadata for functions.

See defining the metadata using DDL, you can define your source function in the VDB as

```
CREATE DATABASE "{vdb-name}";
USE DATABASE "{vdb-name}";
CREATE SERVER AccountsDB FOREIGN DATA WRAPPER oracle OPTIONS ("resource-name" 'java:/oracleDS');
CREATE SCHEMA "{model-name}" SERVER AccountsDB;
```

```
SET SCHEMA "{model-name}";
CREATE FOREIGN FUNCTION SCORE (val integer) RETURNS integer;
```

In an XML VDB:

```
<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"><![CDATA[
           CREATE FOREIGN FUNCTION SCORE (val integer) RETURNS integer;
           .... (other tables, procedures etc)
         ]]>
        </metadata>
    </model>
</vdb>
```

By default when a source can provide metadata, the *Source* model's metadata is automatically retrieved from the source if they were JDBC, File, WebService. The File and WebService sources are static, so one can not add additional metadata on them. However on the JDBC sources you can retrieve the metadata from source and then user can append additional metadata on top of them. For example

```
CREATE DATABASE "{vdb-name}";
USE DATABASE "{vdb-name}";
CREATE SERVER AccountsDB FOREIGN DATA WRAPPER oracle OPTIONS ("resource-name"
'java:/oracleDS');
CREATE SCHEMA "{model-name}" SERVER AccountsDB;
SET SCHEMA "{model-name}";
IMPORT FROM AccountsDB INTO "{model-name}";
CREATE FOREIGN FUNCTION SCORE (val integer) RETURNS integer;
```

In an XML VDB:

```
<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"><![CDATA[
           CREATE FOREIGN FUNCTION SCORE (val integer) RETURNS integer;
         ]]>
        </metadata>
    </model>
</vdb>
```

The above example uses *NATIVE* metadata type (*NATIVE* is the default for source/physical models) first to retrieve schema information from source, then uses *DDL* metadata type to add additional metadata. Only metadata not available via the NATIVE translator logic would need to be specified via DDL.

Alternatively, if you are using custom *MetadataRepository* with your VDB, then provide the "function" metadata directly from your implementation. ex.

```
<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="{metadata-repo-module}"></metadata>
    </model>
</vdb>
```

In the above example, user can implement *MetadataRepository* interface and package the implementation class along with its dependencies in a WildFly module and supply the module name in the above XML. For more information on how to write a Metadata Repository refer to Custom Metadata Repository.

Support for User-Defined Functions (Non-Pushdown)

To define a non-pushdown function, a Java function must be provided that matches the VDB defined metadata. User Defined Function (or UDF) and User Defined Aggregate Function (or UDAF) may be called at runtime just like any other function or aggregate function respectively.

Function Metadata

See User Defined Functions. Make sure you provide the JAVA code implementation details in the properties dialog for the UDF. You can define a UDF or UDAF (User Defined Aggregate Function) as shown below.

```
CREATE DATABASE "{vdb-name}";
USE DATABASE "{vdb-name}";
CREATE VIRTUAL SCHEMA "{model-name}";
SET SCHEMA "{model-name}";
CREATE VIRTUAL FUNCTION celsiusToFahrenheit(celsius decimal) RETURNS decimal OPTIONS (JAVA_CLASS 'org.something
.TempConv',  JAVA_METHOD 'celsiusToFahrenheit');
CREATE VIRTUAL FUNCTION sumAll(arg integer) RETURNS integer OPTIONS (JAVA_CLASS 'org.something.SumAll',  JAVA_M
ETHOD 'addInput', AGGREGATE 'true', VARARGS 'true', "NULL-ON-NULL" 'true');
```

As an XML VDB:

```
<vdb name="{vdb-name}" version="1">
    <model name="{model-name}" type="VIRTUAL">
         <metadata type="DDL"><![CDATA[
             CREATE VIRTUAL FUNCTION celsiusToFahrenheit(celsius decimal) RETURNS decimal OPTIONS (JAVA_CLASS '
org.something.TempConv',  JAVA_METHOD 'celsiusToFahrenheit');
             CREATE VIRTUAL FUNCTION sumAll(arg integer) RETURNS integer OPTIONS (JAVA_CLASS 'org.something.Sum
All',  JAVA_METHOD 'addInput', AGGREGATE 'true', VARARGS 'true', "NULL-ON-NULL" 'true'); ]]> </metadata>
    </model>
</vdb>
```

You must create a Java method that contains the function’s logic. This Java method should accept the necessary arguments, which the Teiid System will pass to it at runtime, and function should return the calculated or altered value.

See DDL Metadata for all possible options related to functions defined via DDL.

Writing the Java Code required by the UDF

The number of input arguments and types must match the function metadata defined in the VDB metadata.

Code Requirements For UDFs

- The java class containing the function method must be defined public.

Note	One implementation class can contain more than one UDF implementation methods.

- The function method must be public and static.

Code Requirements For UDAFs

- The java class containing the function method must be defined public and extend org.teiid.UserDefinedAggregate
- The function method must be public.

Other Considerations

- Any exception can be thrown, but Teiid will rethrow the exception as a `FunctionExecutionException` .
- You may optionally add an additional `org.teiid.CommandContext` argument as the first parameter. The `CommandContext` interface provides access to information about the current command, such as the executing user, Subject, the vdb, the session id, etc. This `CommandContext` parameter should not be declared in the function metadata.

Sample UDF code

```
package org.something;

public class TempConv
{
   /**
   * Converts the given Celsius temperature to Fahrenheit, and returns the
   * value.
   * @param doubleCelsiusTemp
   * @return Fahrenheit
   */
   public static Double celsiusToFahrenheit(Double doubleCelsiusTemp)
   {
      if (doubleCelsiusTemp == null)
      {
         return null;
      }
      return (doubleCelsiusTemp)*9/5 + 32;
   }
}
```

Sample UDAF code

```
package org.something;

public static class SumAll implements UserDefinedAggregate<Integer> {

    private boolean isNull = true;
    private int result;

    public void addInput(Integer... vals) {
        isNull = false;
        for (int i : vals) {
            result += i;
        }
    }

    @Override
    public Integer getResult(org.teiid.CommandContext commandContext) {
        if (isNull) {
            return null;
        }
        return result;
    }

    @Override
    public void reset() {
        isNull = true;
        result = 0;
    }

}
```

Sample CommandContext Usage

```
package org.something;

public class SessionInfo
{
```

```
    /**
    * @param context
    * @return the created Timestamp
    */
    public static Timestamp sessionCreated(CommandContext context)
    {
       return new Timestamp(context.getSession().getCreatedTime());
    }
}
```

The corresponding UDF would be declared as Timestamp sessionCreated().

Post Code Activities

- After coding the functions you should compile the Java code into a Java Archive (JAR) file.

Zip Deployment

The JAR file may be placed in your VDB under the "/lib" directory. It will automatically be used for the VDB classloader classpath when deployed.

AS Module

Create a WildFly module with the JAR file under *<jboss-as>/modules* directory and define the module on the -vdb.xml file as shown below example

```
<vdb name="{vdb-name}" version="1">
    <property name ="lib" value ="{module-name}"></property>
     ...
</vdb>
```

The lib property value may contain a space delimited list of module names if more than one dependency is needed.

Archetype Template UDF Project

One way to start developing a custom user defined function (UDF) is to create a project using the Teiid UDF archetype template. When the project is created from the template, it will create a maven project that contains an example java class and the assembly resources for packaging as a module or a CLI script for configuring via jboss-cli.

Note	The project will be created as an independent project and has no parent maven dependencies. It's designed to be built independent of building Teiid.

You have 2 options for creating a UDF project; in Eclipse by creating a new maven project from the arche type or by using the command line to generate the project.

Create Project in Eclipse

To create a Java project in Eclipse from an arche type, perform the following:

- Open the JAVA perspective
- From the menu select File –> New —> Other
- In the tree, expand Maven and select Maven Project, press Next
- On the "Select project name and Location" window, you can accept the defaults, press Next
- On the "Select an Archetype" window, select Configure button
- Add the remote catalog: link:http://central.maven.org/maven2/ then click OK to return
- Enter "teiid" in the filter to see the Teiid arche types.
- Select the udf-archetype v12.0.0, then press Next
- Enter all the information (i.e., Group ID, Artifact ID, method-name, method-args, return-type etc.) needed to generate the project, then click Finish

The project will be created and name according to the **ArtifactID*.*

Create Project using Command Line

Note	make sure the link:http://central.maven.org/maven2/ repository is accessible via your maven settings.

To create a custom translator project from the command line, you can use the following template command:

- TEMPLATE

```
mvn archetype:generate                              \
  -DarchetypeGroupId=org.teiid.arche-types                \
  -DarchetypeArtifactId=udf-archetype          \
  -DarchetypeVersion=${archetypeVersion}     \
  -DgroupId=${groupId}                      \
  -DartifactId=${udf-artifact-id}   \
  -Dpackage=${package}    \
  -Dversion=0.0.1-SNAPSHOT      \
  -Dudf-name=${functionName}  \
  -Dmethod-name=${methodName}      \
  -Dmethod-args=${methodArguments}   \
  -Dreturn-type=${returnType}      \
```

```
  -Dteiid-version=${teiid-version}
```

- where:

```
  -DarchetypeGroupId    -  is the group ID for the arche type to use to generate
  -DarchetypeArtifactId -  is the artifact ID for the arche type to use to generate
  -DarchetypeVersion    -  is the version for the arche type to use to generate
  -DgroupId     -  (user defined) group ID for the new udf project pom.xml
  -DartifactId      -  (user defined) artifact ID for the new udf project pom.xml
  -Dpackage     -  (user defined) the package structure where the java, module and resource files will be creat
ed
  -Dversion     -  (user defined) the version that the new connector project pom.xml will be
  -Dudf-name        -  (user defined) the name to give the new user defined function, will become the Class Nam
e
  -Dmethod-name         -  (user defined) the name of the method that will be configured in the model procedure
  -Dmethod-args         -  (user defined) the arguments the method will accept.  'Type name[, Type name[,...]]
 Example:  'String arg0' or 'String arg0, integer arg1'
  -Dreturn-type         -  (user defined) the data type of the value returned by the method
  -Dteiid-version   -  the Teiid version the connector will depend upon
```

- EXAMPLE
- this is an example of the template that can be run:

```
mvn archetype:generate                           \
  -DarchetypeGroupId=org.teiid.arche-types               \
  -DarchetypeArtifactId=udf-archetype          \
  -DarchetypeVersion=     \
  -DgroupId=org.example   \
  -DartifactId=udf-function    \
  -Dpackage=org.example.function    \
  -Dversion=0.0.1-SNAPSHOT      \
  -Dudf-name=Function \
  -Dmethod-name=function      \
  -Dmethod-args='String arg1'   \
  -Dreturn-type=String    \
  -Dteiid-version=${teiid-version}
```

When executed, you will be asked to confirm the package property

[INFO] Archetype repository not defined. Using the one from [org.teiid.arche-types:udf-archetype:9.0.1] found in catalog local [INFO] Using property: groupId = org.example [INFO] Using property: artifactId = udf-function [INFO] Using property: version = 0.0.1-SNAPSHOT [INFO] Using property: package = org.example.function [INFO] Using property: method-args = String arg1 [INFO] Using property: method-name = function [INFO] Using property: return-type = String [INFO] Using property: udf-name = Function Confirm properties configuration: groupId: org.example artifactId: udf-function version: 0.0.1-SNAPSHOT package: org.example.function method-args: String arg1 method-name: function return-type: String udf-name: Function Y: : y

```
type Y (yes) and press enter, and the creation of the udf project will be done

Upon creation, a directory based on the _*artifactId*_ will be created, that will
contain the project. 'cd' into that directory and execute a test build to confirm
the project was created correctly:

[source,java]
```

```
mvn clean install
```

```
This should build successfully, and now you are ready to start adding your custom
code.
```

AdminAPI

In most circumstances the admin operations will be performed through the admin console, but it is also possible to invoke admin functionality directly in Java (or a Java scripting language) through the AdminAPI.

All classes for the AdminAPI are in the client jar under the `org.teiid.adminapi` package.

Connecting

An AdminAPI connection, which is represented by the `org.teiid.adminapi.Admin` interface, is obtained through the `org.teiid.adminapi.AdminFactory.createAdmin` methods. `AdminFactory` is a singleton in the teiid-jboss-admin jar, see `AdminFactory.getInstance()`. The `Admin` instance automatically tests its connection and reconnects to a server in the event of a failure. The `close` method should be called to terminate the connection.

See your JBoss installation for the appropriate admin port - the default port is 9999.

Admin Methods

Admin methods exist for monitoring, server administration, and configuration purposes. Note that the objects returned by the monitoring methods, such as getRequests, are read-only and cannot be used to change server state. See the JavaDocs for all of the details

Custom Logging

The Teiid system provides a wealth of information using logging. To control logging level, contexts, and log locations, you should be familiar with container's `standalone.xml` or `domain.xml` configuration file and check out "logging" subsystem. Refer to the Administrator's Guide for more details about different Teiid contexts available.

If you want a custom log handler, follow the directions to write a custom *java.util.logging.Handler*. If you develop a custom logging Handler, the implementation class along should be placed as a jar in "org.jboss.teiid" module and define its name in the module.xml file as part of the module along with any dependencies it may need. See below.

Command Logging API

If you want to build a custom handler for command logging that will have access to java.util.logging LogRecords to the "COMMAND_LOG" context, the handler will receive a instance of LogRecord message, this object will contain a parameter of type `org.teiid.logging.CommandLogMessage`. The relevant Teiid classes are defined in the *teiid-api-14.0.0.jar*. The CommmdLogMessage includes information about vdb, session, command sql, etc. CommandLogMessages are logged at the DEBUG (user queries and source queries on the .SOURCE child context), and TRACE (query plan) levels.

Sample CommandLogMessage Usage

```
package org.something;
import java.util.logging.Handler;
import java.util.logging.LogRecord;

public class CommandHandler extends Handler {
    @Override
    public void publish(LogRecord record) {
    CommandLogMessage msg = (CommandLogMessage)record.getParameters()[0];
        //log to a database, trigger an email, etc.
    }

    @Override
    public void flush() {
    }

    @Override
    public void close() throws SecurityException {
    }
}
```

Audit Logging API

If you want to build a custom handler for command logging that will have access to java.util.logging LogRecords to the "AUDIT_LOG" context, the handler will receive a instance of LogRecord message, this object will contain a parameter of type `org.teiid.logging.AuditMessage`. The `AuditMessage` includes information about user, the action, and the target(s) of the action. The relevant Teiid classes are defined in the *teiid-api-14.0.0.jar*. AuditMessages are logged at the DEBUG level. AuditMessages are used for both data role validation and for logon/logoff events. Only logon events will contain `LogonInfo`.

Sample AuditMessage Usage

```
package org.something;
import java.util.logging.Handler;
import java.util.logging.LogRecord;

public class AuditHandler extends Handler {
    @Override
```

```
    public void publish(LogRecord record) {
    AuditMessage msg = (AuditMessage)record.getParameters()[0];
        //log to a database, trigger an email, etc.
    }

    @Override
    public void flush() {
    }

    @Override
    public void close() throws SecurityException {
    }
}
```

Configuration

Now that you have developed a custom handler class, now package implementation in Jar file, then copy this Jar file into `<jboss-as7>/modules/org/jboss/teiid/main` folder, and edit `module.xml` file in the same directory and add

```
<resource-root path="{your-jar-name}.jar" />
```

then use the cli to update the logging configuration, such as shown with the auditcommand scripts in the bin/scripts directory or edit `standalone-teiid.xml` or `domain.xml` file by locating the "logging" subsystem and add the following entries:

```
<custom-handler name="COMMAND" class="org.teiid.logging.CommandHandler"
  module="org.jboss.teiid">
</custom-handler>

..other entries

<logger category="org.teiid.COMMAND_LOG">
    <level name="DEBUG"/>
    <handlers>
        <handler name="COMMAND"/>
    </handlers>
</logger>
```

Change the above configuration accordingly for AuditHandler, if you are working with Audit Messages.

Runtime Updates

Teiid supports several mechanisms for updating the runtime system.

Data Updates

Data change events are used by Teiid to invalidate result set cache entries. Result set cache entries are tracked by the tables that contributed to their results. By default Teiid will capture internal data events against physical sources and distribute them across the cluster. This approach has several limitations. First updates are scoped only to their originating VDB/version. Second updates made out side of Teiid are not captured. To increase data consistency external change data capture tools can be used to send events to Teiid. From within a Teiid cluster the `org.teiid.events.EventDistributorFactory` and `org.teiid.events.EventDistributor` can be used to distribute change events. The `EventDistributorFactory` can be looked up by its name *"teiid/event-distributor-factory"*. See Programmatic Control for a dataModification example.

When externally capturing all update events, *"detect-change-events"* property in the teiid subsystem in can be set to *false*, to not duplicate change events. By default, this property is set to *true*.

Note	Using the `org.teiid.events.EventDistributor` interface you can also update runtime metadata. Please check the API.

The use of the other `EventDistributor` methods to manually distribute other events is not always necessary. Check the System Procedures for SQL based updates.

Runtime Metadata Updates

Runtime updates via system procedures and DDL statements are by default ephemeral. They are effective across the cluster only for the currently running vdbs. With the next vdb start the values will revert to whatever is stored in the vdb. Updates may be made persistent though by configuring a `org.teiid.metadata.MetadataRepository`. An instance of a `MetadataRepository` can be installed via the vdb.xml file in the META-INF directory or use a VDB file as below.

```
<vdb name="{vdb-name}" version="1">
    <model name="{model-name}" type="VIRTUAL">
         <metadata type="{jboss-as-module-name}"></metadata>
    </model>
</vdb>
```

In the above code fragment, replace the {jboss-as-module-name} with a WildFly module name that has library that implements the `org.teiid.metadata.MetadataRepository` interface and defines file "META-INF/services/org.teiid.metadata.MetadataRepository" with name of the implementation file.

The `MetadataRepository` repository instance may fully implement as many of the methods as needed and return null from any unneeded getter.

It is not recommended to directly manipulate `org.teiid.metadata.AbstractMetadataRecord` instances. System procedures and DDL statements should be used instead since the effects will be distributed through the cluster and will not introduce inconsistencies.

`org.teiid.metadata.AbstractMetadataRecord` objects passed to the `MetadataRepository` have not yet been modified. If the `MetadataRepository` cannot persist the update, then a `RuntimeException` should be thrown to prevent the update from being applied by the runtime engine.

The MetadataRepository can be accessed by multiple threads both during load or at runtime with through DDL statements. Your implementation should handle any needed synchronization.

Costing Updates

See the Reference for the system procedures `SYSADMIN.setColumnStats` and `SYSADMIN.setTableStats`. To make costing updates persistent `MetadataRepository` implementations should be provided for:

```
TableStats getTableStats(String vdbName, String vdbVersion, Table table);
void setTableStats(String vdbName, String vdbVersion, Table table, TableStats tableStats);
ColumnStats getColumnStats(String vdbName, String vdbVersion, Column column);
void setColumnStats(String vdbName, String vdbVersion, Column column, ColumnStats columnStats);
```

Schema Updates

See the Reference for supported DDL statements. To make schema updates persistent implementations should be provided for:

```
String getViewDefinition(String vdbName, String vdbVersion, Table table);
void setViewDefinition(String vdbName, String vdbVersion, Table table, String viewDefinition);
String getInsteadOfTriggerDefinition(String vdbName, String vdbVersion, Table table, Table.TriggerEvent triggerOperation);
void setInsteadOfTriggerDefinition(String vdbName, String vdbVersion, Table table, Table.TriggerEvent triggerOperation, String triggerDefinition);
boolean isInsteadOfTriggerEnabled(String vdbName, String vdbVersion, Table table, Table.TriggerEvent triggerOperation);
void setInsteadOfTriggerEnabled(String vdbName, String vdbVersion, Table table, Table.TriggerEvent triggerOperation, boolean enabled);
String getProcedureDefinition(String vdbName, String vdbVersion, Procedure procedure);
void setProcedureDefinition(String vdbName, String vdbVersion, Procedure procedure, String procedureDefinition);
LinkedHashMap<String, String> getProperties(String vdbName, String vdbVersion, AbstractMetadataRecord record);
void setProperty(String vdbName, String vdbVersion, AbstractMetadataRecord record, String name, String value);
```

Custom Metadata Repository

If the provided metadata facilities are not sufficient then a developer can extend the *MetadataRepository* class provided in the *org.teiid.api* jar to plug-in their own metadata facilities into the Teiid engine. For example, a user can write a metadata facility that is based on reading data from a database or a JCR repository.

See the arche-type for creating a custom metadata repository.

Or see setting up the build environment to start development. For Example:

Sample Java Code

```
import org.teiid.metadata.MetadataRepository;
...

package com.something;

public class CustomMetadataRepository implements MetadataRepository {
    @Override
    public void loadMetadata(MetadataFactory factory, ExecutionFactory executionFactory, Object connectionFacto
ry)
        throws TranslatorException {
        /* Provide implementation and fill the details in factory */
        ...
    }
}
```

Then build a JAR archive with above implementation class and create file a named *org.teiid.metadata.MetadataRepository* in the *META-INF/services* directory with contents:

```
com.something.CustomMetadataRepository
```

Once the JAR file has been built, it needs to be deployed in the WildFly as a module under *<jboss-as>/modules* directory. Follow the below steps to create a module.

- Create a directory *<jboss-as>/modules/com/something/main*
- Under this directory create a "module.xml" file that looks like

Sample module.xml file

```
<?xml version="1.0" encoding="UTF-8"?>
<module xmlns="urn:jboss:module:1.0" name="com.something">
    <resources>
        <resource-root path="something.jar" />
    </resources>
    <dependencies>
        <module name="javax.api"/>
        <module name="org.teiid.common-core"/>
        <module name="org.teiid.teiid-api" />
    </dependencies>
</module>
```

- Copy the jar file under this same directory. Make sure you add any additional dependencies if required by your implementation class under dependencies.
- Restart the server

The below XML fragment shows how to configure the VDB with the custom metadata repository created

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="{metadata-repo-module}"></metadata>
    </model>
</vdb>
```

Now when this VDB gets deployed, it will call the *CustomMetadataRepository* instance for metadata of the model. Using this you can define metadata for single model or for the whole VDB pragmatically. Be careful about holding state and synchronization in your repository instance.

Development Considerations

- `MetadataRepository` instances are created on a per vdb basis and may be called concurrently for the load of multiple models.
- See the `MetadataFactory` and the `org.teiid.metadata` package javadocs for metadata construction methods and objects. For example if you use your own DDL, then call the `MetadataFactory.parse(Reader)` method. If you need access to files in a VDB zip deployment, then use the `MetadataFactory.getVDBResources` method.
- Use the `MetadataFactory.addPermission` and add `MetadataFactory.addColumnPermission` method to grant permissions on the given metadata objects to the named roles. The roles should be declared in your vdb.xml, which is also where they are typically tied to container roles.

PreParser

If it is desirable to manipulate incoming queries prior to being handled by Teiid logic, then a custom pre-parser can be installed.

A PreParser may be set at a global level for all VDBs, or at a per VDB level. If both are specified the global PreParser will be called first, then the per VDB PreParser.

Use the *PreParser* interface provided in the *org.teiid.api* jar to plug-in a pre-parser for the Teiid engine. See Setting up the build environment to start development. For Example:

Sample Java Code

```
import org.teiid.PreParser;
...

package com.something;

public class CustomPreParser implements PreParser {

    @Override
    public String preParse(String command, CommandContext context) {
        //manipulate the command
    }
}
```

If this is intended to be a global PreParser, then create a file named *org.teiid.PreParser* in *META-INF/services* directory with contents:

```
com.something.CustomPreParser
```

After the jar has been built, it needs to be deployed in the WildFly as a module under *<jboss-as>/modules* directory. Follow the below steps to create a module.

- Create a directory *<jboss-as>/modules/com/something/main*
- Under this directory create a `module.xml` file that looks like

Sample module.xml file

```
<?xml version="1.0" encoding="UTF-8"?>
<module xmlns="urn:jboss:module:1.0" name="com.something">
    <resources>
        <resource-root path="something.jar" />
    </resources>
    <dependencies>
        <module name="javax.api"/>
        <module name="javax.resource.api"/>
        <module name="org.teiid.common-core"/>
        <module name="org.teiid.teiid.api" />
    </dependencies>
</module>
```

- Copy the jar file under this same directory. Make sure you add any additional dependencies if required by your implementation class under dependencies.
- If this is a global PreParser, then use the cli or modify the configuration to set the preparser-module in the Teiid subsystem configuration to the appropriate module name.

- If this is a per VDB PreParser, then update the vdb property "preparser-class" to be the class name of your PreParser. The VDB class path also needs to be updated to include the PreParser module, which can be done by adding the module name to the "lib" property.

Sample vdb.xml properties

```
<vdb name="..." version="...">
   <property name="lib" value="preparser-module-name"/>
   <property name="preparser-class" value="com.something.CustomPreParser"/>
   ...
</vdb>
```

- Restart the server for the module to become available.

Development Considerations

- Changing the incoming query to a different type of statement is not recommended as are any modifications to the number or types of projected symbols.
- When using Teiid Embedded you just need to include the jar with the PreParser in the application class path - as modules are not used.

Archetype Template PreParser Project

One way to start developing a custom preparser is to create a project using the Teiid archetype template. When the project is created from the template, it will contain an example class and resources for you to begin adding your custom logic. Additionally, the maven dependencies are defined in the pom.xml so that you can begin compiling the classes.

Note	The project will be created as an independent project and has no parent maven dependencies. It’s designed to be built independent of building Teiid.

You have 2 options for creating a translator project; in Eclipse by creating a new maven project from the arche type or by using the command line to generate the project.

Create Project in Eclipse

To create a Java project in Eclipse from an arche type, perform the following:

- Open the JAVA perspective
- From the menu select File –> New —> Other
- In the tree, expand Maven and select Maven Project, press Next
- On the "Select project name and Location" window, you can accept the defaults, press Next
- On the "Select an Archetype" window, select Configure button
- Add the remote catalog: link:http://central.maven.org/maven2/ then click OK to return
- Enter "teiid" in the filter to see the Teiid arche types.
- Select the preparser-archetype, then press Next
- Enter all the information (i.e., Group ID, Artifact ID, etc.) needed to generate the project, then click Finish

The project will be created and name according to the **ArtifactID**.

Create Project using Command Line

Note	make sure the link:http://central.maven.org/maven2/ repository is accessible via your maven settings.

To create a custom preparser project from the command line, you can use the following template command:

- TEMPLATE

```
mvn archetype:generate           \
  -DarchetypeGroupId=org.teiid.arche-types               \
  -DarchetypeArtifactId=preparser-archetype          \
  -DarchetypeVersion=${archetypeVersion}     \
  -DgroupId=${groupId}                  \
  -DartifactId=${preparser-artifact-id} \
  -Dpackage=${package}    \
  -Dversion=0.0.1-SNAPSHOT      \
  -Dclass-name=${className}     \
  -Dteiid-version=${teiidVersion}
```

- where:

```
  -DarchetypeGroupId    -  is the group ID for the arche type to use to generate
  -DarchetypeArtifactId -  is the artifact ID for the arche type to use to generate
  -DarchetypeVersion    -  is the version for the arche type to use to generate
  -DgroupId     -  (user defined) group ID for the new preparser project pom.xml
  -DartifactId      -  (user defined) artifact ID for the new example project pom.xml
  -Dpackage     -  (user defined) the package structure where the java, module and resource files will be creat
ed
  -Dversion     -  (user defined) the version that the new connector project pom.xml will be
  -Dclass-name      -  (user defined) the class name to give the new user preparser, will become the Class Name
  -Dteiid-version   -  the Teiid version the connector will depend upon
```

- EXAMPLE
- this is an example of the template that can be run:

```
mvn archetype:generate             \
  -DarchetypeGroupId=org.teiid.arche-types               \
  -DarchetypeArtifactId=preparser-archetype          \
  -DarchetypeVersion=12.0.0    \
  -DgroupId=org.example   \
  -DartifactId=preparser-mypreparser    \
  -Dpackage=org.example.mypreparser    \
  -Dversion=0.0.1-SNAPSHOT      \
  -Dclass-name=MyPreParser      \
  -Dteiid-version=14.0.0
```

When executed, you will be asked to confirm the package property

```
[INFO] Using property: groupId = org.example
[INFO] Using property: artifactId = preparser-mypreparser
[INFO] Using property: version = 0.0.1-SNAPSHOT
[INFO] Using property: package = org.example.mypreparser
[INFO] Using property: class-name = MyPreParser
[INFO] Using property: teiid-version = 14.0.0
Confirm properties configuration:
groupId: org.teiid.preparser
artifactId: preparser-myParser
version: 0.0.1-SNAPSHOT
package: org.example.mypreparser
class-name: MyPreParser
teiid-version: 14.0.0
 Y: : y
```

type Y (yes) and press enter, and the creation of the preparser project will be done

Upon creation, a directory based on the **artifactId* will be created, that will contain the project. 'cd' into that directory and execute a test build to confirm the project was created correctly:

```
 mvn clean install
```

This should build successfully, and now you are ready to start adding your custom code.

Embedded Guide

Embedded is a light-weight version of Teiid for use in any Java 8+ JRE. WildFly nor any application server is not required. This feature/kit are still evolving. Please consult the source examples and even unit tests utilizing the EmbeddedServer for a more complete guide as to its use.

Table of Contents

Configuration

The primary way to configure Teiid Embedded is with the `EmbeddedConfiguration` class. It is provided to the `EmbeddedServer` at start-up and dictates much of the behavior of the embedded instance. From there the running server instance may have translators and VDBs deployed as needed. Additional modifications to the `EmbeddedConfiguration` after the server is started will not have an effect.

In many cases an `EmbeddedConfiguration` instance can just be instantiated and passed to the `EmbeddedServer` without the need to set additional properties. Many properties, including those used to configure the BufferManager, will be given a similar name to their server side counter part - for example setProcessorBatchSize.

Important	Most of the default configuration values for memory and threads assume that there is only one Teiid instance in the vm. If you are using multiple Teiid Embedded instances in the same vm, then memory and thread resources should be configured manually.

The Classpath

Embedded Using Maven

Your application is responsible for having the appropriate classpath to utilize Teiid embedded. Typically you will want all transitive dependencies from referenced Teiid artifacts to be included. Optional dependencies, such as Hibernate core, will be needed for specific features - such as utilizing the JDBC translator support for dependent joins using temp tables.

Note	With Teiid 10+ the maven coordinate group for most Teiid artifacts changed from org.jboss.teiid to just org.teiid. Please update your pom files accordingly.

Some of the Teiid transitive dependencies have known vulnerabilities. WildFly/Teiid addresses this by introducing managed dependency overrides. It is recommended that you include these overrides in your usage of Teiid Embedded by importing the Teiid parent pom in your dependency management section:

```
<dependencyManagement>
```

```
  <dependencies>
    <dependency>
      <groupId>org.teiid</groupId>
      <artifactId>teiid-parent</artifactId>
      <version>${version.teiid}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
```

Dependencies

If you are trying run Teiid Embedded as a Maven based project, the `runtime` , `admin` , `connector` , `translator` dependencies necessary are

```
<dependency>
    <groupId>org.teiid</groupId>
    <artifactId>teiid-runtime</artifactId>
</dependency>

<dependency>
    <groupId>org.teiid</groupId>
    <artifactId>teiid-admin</artifactId>
</dependency>

<dependency>
    <groupId>org.teiid.connectors</groupId>
    <artifactId>translator-SOURCE</artifactId>
</dependency>

<dependency>
    <groupId>org.teiid.wildfly.connectors</groupId>
    <artifactId>connector-SOURCE</artifactId>
</dependency>
```

You would include all translator/connectors needed by your project.

Optional Libraries

If you include a dependency to org.teiid:teiid-data-quality, the osdq data quality functions will be available for use with Embedded.

If you include a dependency to org.teiid:cache-infinispan, Infinispan will be used for caching.

If you do not need XML type support including XPath and SQL/XML functions like XMLTABLE, then you may also choose to exclude saxon, xom, and nux from usage by the runtime by using excludes:

```
<dependency>
    <groupId>org.teiid</groupId>
    <artifactId>teiid-runtime</artifactId>
    <exclusions>
        <exclusion>
            <groupId>org.teiid</groupId>
            <artifactId>teiid-optional-xml</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

Some geospatial support requires additional dependencies. If you need no or minimal support (no geojson nor projection), then you may also choose to exclude this from the runtime by using excludes:

```
<dependency>
```

```
    <groupId>org.teiid</groupId>
    <artifactId>teiid-runtime</artifactId>
    <exclusions>
        <exclusion>
            <groupId>org.teiid</groupId>
            <artifactId>teiid-optional-geo</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

Some json support requires additional dependencies. If you need no or minimal support (no jsonpath support), then you may also choose to exclude this from the runtime by using excludes:

```
<dependency>
    <groupId>org.teiid</groupId>
    <artifactId>teiid-runtime</artifactId>
    <exclusions>
        <exclusion>
            <groupId>org.teiid</groupId>
            <artifactId>teiid-optional-json</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

VDB Deployment

VDBs may be deployed in several ways in Embedded.

VDB Metadata API

VDB deployment can be done directly through VDB metadata objects that are the underpinning of vdb.xml deployment. Models (schemas) are deployed as a set to form a named vdb - see the `EmbeddedServer.deployVDB` method.

XML Deployment

Similar to a server based -vdb.xml deployment an `InputStream` may be given to a vdb.xml file - see the `EmbeddedServer.deployVDB(InputStream)` method.

Zip Deployment

Similar to a server based .vdb deployment a `URL` may be given to a zip file - see the `EmbeddedServer.deployVDBZip` method. The use of the zip lib for dependency loading is not enabled in Embedded.

See VDB Guide and Metadata Repositories for more on a typical vdb file and zip structures.

Support Teiid Designer 7 and later VDBs is deprecated and are subject to all of the limitations/differences highlighted in this guide. To use a Teiid Designer VDB requires including the teiid-metadata dependency:

```
<dependency>
    <groupId>org.teiid</groupId>
    <artifactId>teiid-metadata</artifactId>
</dependency>
```

Translators

Translators instances can be scoped to a VDB in AS using declarations in a vdb.xml file, however named instances in embedded are scoped to the entire `EmbeddedServer` and must be registered via the `EmbeddedServer.addTranslator` methods. Note that there are three `addTranslator` methods:

- `addTranslator(Class<? extends ExecutionFactory> clazz)` - Adds a default instance of the ExecutionFactory, using the default name either from the Translator annotation or the class name.
- `addTranslator(String name, ExecutionFactory<?, ?> ef)` - Adds a pre-initialized (ExecutionFactory.start() must have already been called) instance of the ExecutionFactory, using the given translator name. The instance will be shared for all usage.
- `addTranslator(String name, String type, Map<String, String> properties)` - Adds a definition of an override translator - this is functionally equivalent to using a vdb.xml translator override.

A new server instance does not assume any translators are deployed and does not perform any sort of library scanning to find translators.

Sources

The Embedded Server will still attempt to lookup the given JNDI connection factory names via JNDI. In most non-container environments it is likely that no such bindings exist. In this case the Embedded Server instance must have `ConnectionFactoryProvider` instances manually registered, either using the `EmbeddedServer.addConnectionFactory` method, or the `EmbeddedServer.addConnectionFactoryProvider` method to implement `ConnectionFactoryProvider` registering. Note that the Embedded Server does not have built-in pooling logic, so to make better use of a standard `java.sql.DataSource` or to enable proper use of `javax.sql.XADataSource` you must first configure the instance via a third-party connection pool.

Example - Deployment

```
EmbeddedServer es = new EmbeddedServer();
EmbeddedConfiguration ec = new EmbeddedConfiguration();
//set any configuration properties
ec.setUseDisk(false);
es.start(ec);

//example of adding a translator by pre-initialized ExecutionFactory and given translator name
H2ExecutionFactory ef = new H2ExecutionFactory()
ef.setSupportsDirectQueryProcedure(true);
ef.start();
es.addTranslator("translator-h2", ef);

//add a Connection Factory with a third-party connection pool
DataSource ds = EmbeddedHelper.newDataSource("org.h2.Driver", "jdbc:h2:mem://localhost/~/account", "sa", "sa");
es.addConnectionFactory("java:/accounts-ds", ds);

//add a vdb

//physical model
ModelMetaData mmd = new ModelMetaData();
mmd.setName("my-schema");
mmd.addSourceMapping("my-schema", "translator-h2", "java:/accounts-ds");

//virtual model
ModelMetaData mmd1 = new ModelMetaData();
mmd1.setName("virt");
mmd1.setModelType(Type.VIRTUAL);
mmd1.setSchemaSourceType("ddl");
mmd1.setSchemaText("create view \"my-view\" OPTIONS (UPDATABLE 'true') as select * from \"my-table\"");

es.deployVDB("test", mmd, mmd1);
```

Secured Data Sources

If Source related security authentication, for example, if you want connect/federate/integrate Twitter supplied rest source, a security authentication is a necessary, the following steps can use to execute security authentication:

1. refer to Secure Embedded with PicketBox start section to develop a SubjectFactory,

2. initialize a ConnectionManager with ironjacamar libaries, set SubjectFactory to ConnectionManager

3. use the following method to create ConnectionFactory

Example - Secured Data Sources

```
WSManagedConnectionFactory mcf = new WSManagedConnectionFactory();
NoTxConnectionManagerImpl cm = new NoTxConnectionManagerImpl();
cm.setSecurityDomain(securityDomain);
cm.setSubjectFactory(new EmbeddedSecuritySubjectFactory(authConf))
Object connectionFactory = mcf.createConnectionFactory(cm);
server.addConnectionFactory("java:/twitterDS", connectionFactory);
```

twitter-as-a-datasource is a completed example.

Access from client applications

Typically when Teiid is deployed as Embedded Server, and if your end user application is also deployed in the same virtual machine as the Teiid Embedded, you can use **Local JDBC Connection**, to access to your virtual database. For example:

Example - Local JDBC Connection

```
EmbeddedServer es = ...
Driver driver = es.getDriver();
Connection conn = driver.connect("jdbc:teiid:<vdb-name>", null);
// do work with conn; create statement and execute it
conn.close();
```

This is the most efficient method as it does not impose any serialization of objects.

If your client application is deployed in remote VM, or your client application is not a JAVA based application then accesses to the Teiid Embedded is not possible through above mechanism. In those situations, you need to open a socket based connection from remote client application to the Embedded Teiid Server. By default, when you start the Embedded Teiid Sever it does not add any capabilities to accept remote JDBC/ODBC based connections. If you would like to expose the functionality to accept remote JDBC/ODBC connection requests, then configure necessary **transports** during the initialization of the Teiid Embedded Server. The example below shows a sample code to enable a ODBC transport

Example - Remote ODBC transport

```
EmbeddedServer es = new EmbeddedServer()
SocketConfiguration s = new SocketConfiguration();
s.setBindAddress("<host-name>");
s.setPortNumber(35432);
s.setProtocol(WireProtocol.pg);
EmbeddedConfiguration config = new EmbeddedConfiguration();
config.addTransport(s);
es.start(config);
```

Example - SSL transport

```
EmbeddedServer server = new EmbeddedServer();
...
EmbeddedConfiguration config = new EmbeddedConfiguration();
SocketConfiguration socketConfiguration = new SocketConfiguration();

SSLConfiguration sslConfiguration = new SSLConfiguration();

//Settings shown with their default values
//sslConfiguration.setMode(SSLConfiguration.ENABLED);
//sslConfiguration.setAuthenticationMode(SSLConfiguration.ONEWAY);
//sslConfiguration.setSslProtocol(SocketUtil.DEFAULT_PROTOCOL);
//sslConfiguration.setKeymanagementAlgorithm(KeyManagerFactory.getDefaultAlgorithm());
```

```
//optionally restrict the cipher suites
//sslConfiguration.setEnabledCipherSuites("SSL_RSA_WITH_RC4_128_MD5,SSL_RSA_WITH_RC4_128_SHA");

//for the server key
sslConfiguration.setKeystoreFilename("ssl-example.keystore");
sslConfiguration.setKeystorePassword("redhat");
sslConfiguration.setKeystoreType("JKS");
sslConfiguration.setKeystoreKeyAlias("teiid");
sslConfiguration.setKeystoreKeyPassword("redhat");

//for two way ssl set a truststore for client certs
//sslConfiguration.setTruststoreFilename("ssl-example.truststore");
//sslConfiguration.setTruststorePassword("redhat");

socketConfiguration.setSSLConfiguration(sslConfiguration);
config.addTransport(socketConfiguration);

server.start(config);
```

if you want to add a JDBC transport, follow the instructions above, however set the protocol to `WireProtocol.teiid` and choose a different port number. Once the above server is running, you can use same instructions as Teiid Server to access Embedded Teiid Server from remote client application. Note that you can add multiple transports to single Embedded Server instance, to expose different transports.

Security

The primary interface for Teiid embedded's security is the `org.teiid.security.SecurityHelper` in the engine jar. The SecurityHelper instance is associated with with the EmbeddedServer via `EmbeddedConfiguration.setSecurityHelper` . If no SecurityHelper is set, then no authentication will be performed. A SecurityHelper controls authentication and associates a security context with a thread. How a security context is obtained can depend upon the security domain name. The default security domain name is `teiid-security` and can be changed via `EmbeddedConfiguration.setSecurityDomain` . The effective security domain may also be configured via a transport of the VDB.

See the JBoss Security Helper source for an example of expected mechanics.

You can just return null from negotiateGssLogin unless you want to all GSS authentications from JDBC/ODBC.

Example

embedded-portfolio-security demonstrates how to implement security authentication in Teiid Embedded:

- EmbeddedSecurityHelper is the implementation of `org.teiid.security.SecurityHelper`
- users.properties and roles.properties in class path user to pre define users and roles
- application-policy's name in authentication.conf should match to security domain(`EmbeddedConfiguration.setSecurityDomain`)

Transactions

Transaction processing requires setting the `TransactionManager` in the `EmbeddedConfiguration` used to start the `EmbeddedServer` . A client facing `javax.sql.DataSource` is not provided for embedded. However the usage of provided `java.sql.Driver` should be sufficient as the embedded server is by default able to detect thread bound transactions and appropriately propagate the transaction to threads launched as part of request processing. The usage of local connections is also permitted.

AdminApi

Embedded provides a the `Admin` interface via the `EmbeddedServer.getAdmin` method. Not all methods are implemented for embedded - for example those that deal with data sources. Also the deploy method may only deploy VDB xml artifacts.

Logging

Teiid by default use JBoss Logging, which will utilize JUL (Java Util Logging) or other common logging frameworks depending upon their presence in the classpath. Refer to Logging in Teiid Embedded for details.

The internal interface for Teiid embedded's logging is `org.teiid.logging.Logger` in teiid-api jar. The Logger instance is associated with the `org.teiid.logging.LogManager` via static method `LogManager.setLogListener()`. You may alternatively choose to directly set a `Logger` of your choice.

Other Differences Between Teiid Embedded and an AS Deployment

- There is no default JDBC/ODBC socket transport in embedded. You are expected to obtain a `Driver` connection via the `EmbeddedServer.getDriver` method. If you want remote JDBC/ODBC transport see above on how to add a transport.
- A `MetadataRepository` is scoped to a VDB in AS, but is scoped to the entire `EmbeddedServer` instance and must be registered via the `EmbeddedServer.addMetadataRepository` method.
- MDC logging values are not available as Java logging lacks the concept of a mapped diagnostic context.
- Translator overrides in vdb.xml files is not supported, but you may add overridden translators using the addTranslator methods that accept an ExecutionFactory instance or a property set.
- The default for the maximum disk space used by the buffer manager is 5 GB, rather than 50 GB.
- VDB imports are processed only at deployment time. A missing vdb import results in a failed deployment. If the imported vdb is redployed after the importing vdb is deployed, the importing vdb is not redeployed.

Logging in Teiid Embedded

Teiid's LogManager is an interface to a single logging framework that is easily accessible by any component. Using the LogManager, a component can quickly submit a log message, and can rely upon the LogManager to determine

- whether that message is to be recorded or discarded
- where to send any recorded messages

JBoss Logging

JBoss Logging is used by default. The JBoss Logging jar is already in the kit and you just need to ensure the jboss-logging library is in your class path. If you use Maven, add the dependency as shown below:

```
<dependency>
    <groupId>org.jboss.logging</groupId>
    <artifactId>jboss-loggging</artifactId>
</dependency>
```

Bridging with JBoss Logging

JBoss LogManager is a replacement for the JDK logging system LogManager that fixes or works around many serious problems in the default implementation. To use JBoss LogManager with JBoss Logging, the only need to do is add jboss-logmanager library to class path. If use Maven to pull dependencies, add the dependency as shown below:

```
<dependency>
    <groupId>org.jboss.logging</groupId>
    <artifactId>jboss-logmanager</artifactId>
</dependency>
```

TeiidEmbeddedLogging is a example for Logging with JBoss LogManager.

A sample logging.properties for Teiid Embedded:

```
loggers=sun.rmi,com.arjuna

logger.level=TRACE
logger.handlers=FILE,CONSOLE

logger.sun.rmi.level=WARN
logger.sun.rmi.useParentHandlers=true

logger.com.arjuna.level=WARN
logger.com.arjuna.useParentHandlers=true

handler.CONSOLE=org.jboss.logmanager.handlers.ConsoleHandler
handler.CONSOLE.level=INFO
handler.CONSOLE.formatter=COLOR-PATTERN
handler.CONSOLE.properties=autoFlush,target,enabled
handler.CONSOLE.autoFlush=true
```

```
handler.CONSOLE.target=SYSTEM_OUT
handler.CONSOLE.enabled=true

handler.FILE=org.jboss.logmanager.handlers.PeriodicRotatingFileHandler
handler.FILE.formatter=PATTERN
handler.FILE.properties=append,autoFlush,enabled,suffix,fileName
handler.FILE.constructorProperties=fileName,append
handler.FILE.append=true
handler.FILE.autoFlush=true
handler.FILE.enabled=true
handler.FILE.suffix=.yyyy-MM-dd
handler.FILE.fileName=target/teiid-embedded.log

formatter.PATTERN=org.jboss.logmanager.formatters.PatternFormatter
formatter.PATTERN.properties=pattern
formatter.PATTERN.pattern=%d{yyyy-MM-dd HH\:mm\:ss,SSS} %-5p \[%c\] (%t) %s%e%n

formatter.COLOR-PATTERN=org.jboss.logmanager.formatters.PatternFormatter
formatter.COLOR-PATTERN.properties=pattern
formatter.COLOR-PATTERN.pattern=%K{level}%d{HH\:mm\:ss,SSS} %-5p \[%c\] (%t) %s%e%n
```

Bridging with Log4j

To bridge JBoss Logging with Log4j, the only need to do is have a 1.x log4j jar in your class path.

If your system use Log4j as logging framework, with above JBoss LogManager bridge Log4j functionality and steps in Bridging with JBoss Logging, it’s easy to set up logging framework consistent between Teiid Embedded and your system.

Secure Embedded with PicketBox

Secure Embedded with PicketBox.

Table of Contents

Steps of implement a JAAS authentication

PicketBox is a Java Security Framework that build on top of JAAS. PicketBox is configured via a schema formatted Security Configuration File(security-config_5_0.xsd) and provides various LoginModule Implementations (UsersRolesLoginModule, LdapExtLoginModule, DatabaseServerLoginModule, etc). The following are 5 key steps to execute a authentication:

```
//1. establish the JAAS Configuration with picketbox authentication xml file
SecurityFactory.prepare();

//2. load picketbox authentication xml file
PicketBoxConfiguration config = new PicketBoxConfiguration();
config.load(SampleMain.class.getClassLoader().getResourceAsStream("picketbox/authentication.conf"));

//3. get AuthenticationManager
AuthenticationManager authManager = SecurityFactory.getAuthenticationManager(securityDomain);

//4. execute authentication
authManager.isValid(userPrincipal, credString, subject);

//5. release resource
SecurityFactory.release();
```

Teiid Embedded exposes 2 methods for security authentication:

- EmbeddedConfiguration.setSecurityHelper() - associated with a *org.teiid.security.SecurityHelper* in the engine jar. If no SecurityHelper is set, then no authentication will be performed.
- EmbeddedConfiguration.setSecurityDomain() - associated with a application-policy's name in Security Configuration file. If no SecurityDomain is set, then a default `teiid-security` will be used.

EmbeddedSecurityHelper is a sample implementation of *SecurityHelper*, authentication.conf is a sample Security Configuration file.

How to develop a SecurityHelper

Add 'teiid-engine-VERSION.jar'to classpath is necessary. If you are using the maven to pull artifacts, the engine dependency can added as below,

```
<dependency>
    <groupId>org.teiid</groupId>
    <artifactId>teiid-engine</artifactId>
</dependency>
```

The key to develop a SecurityHelper is implement the authenticate() method. PicketBox's 5 key steps to execute an authentication which depicted in Steps of implement a JAAS authentication is shown in the example below:

```
@Override
public SecurityContext authenticate(String securityDomain, String baseUserName, Credentials credentials, String
 applicationName) throws LoginException {

    SecurityFactory.prepare();
    try {
        PicketBoxConfiguration config = new PicketBoxConfiguration();
        config.load(this.getClass().getClassLoader().getResourceAsStream("picketbox/authentication.conf"));

        AuthenticationManager authManager = SecurityFactory.getAuthenticationManager(securityDomain);
        if (authManager != null){
            final Principal userPrincipal = new SimplePrincipal(baseUserName);
            final Subject subject = new Subject();
            final String credString = credentials==null?null:new String(credentials.getCredentialsAsCharArray()
);
            final String domain = securityDomain;
            boolean isValid = authManager.isValid(userPrincipal, credString, subject);
            if (isValid) {
                SecurityContext securityContext = AccessController.doPrivileged(new PrivilegedAction<SecurityCo
ntext>(){
                    @Override
                    public SecurityContext run() {
                        SecurityContext sc;
                        try {
                            sc = SecurityContextFactory.createSecurityContext(userPrincipal, credString, subjec
t, domain);
                        } catch (Exception e) {
                            throw new RuntimeException(e);
                        }
                        return sc;
                    }});
                return securityContext;
            }
        }
    } finally {
        SecurityFactory.release();
    }
    throw new LoginException("The username " +  baseUserName + " and/or password could not be authenticated by
security domain " + securityDomain + ".");
}
```

You can just return null from negotiateGssLogin unless you want to all GSS authentications from JDBC/ODBC.

Embedded Security with UsersRolesLoginModule

Add the following content to PicketBox Security Configuration file:

```
<application-policy name = "teiid-security">
    <authentication>
        <login-module code = "org.jboss.security.auth.spi.UsersRolesLoginModule" flag = "required"></login-modu
le>
    </authentication>
</application-policy>
```

To prepare users/roles by add users.properties and roles.properties to class path. A sample of users.properties

```
testUser=password
```

A sample of roles.properties

```
testUser=user
```

To start Embedded Server with UsersRolesLoginModule based security authentication via:

```
EmbeddedServer server =
...
EmbeddedConfiguration config = new EmbeddedConfiguration();
config.setSecurityDomain("teiid-security-file");
config.setSecurityHelper(new EmbeddedSecurityHelper());
server.start(config);
```

Embedded Security with LdapExtLoginModule

Add the following content to the PicketBox Security Configuration File:

```
<application-policy name = "teiid-security-ldap">
    <authentication>
        <login-module code = "org.jboss.security.auth.spi.LdapExtLoginModule" flag = "required">
            <module-option name="java.naming.factory.initial">com.sun.jndi.ldap.LdapCtxFactory</module-option>
            <module-option name="java.naming.provider.url">ldap://HOST:389</module-option>
            <module-option name="java.naming.security.authentication">simple</module-option>
            <module-option name="bindDN">cn=Manager,dc=example,dc=com</module-option>
            <module-option name="bindCredential">redhat</module-option>
            <module-option name="baseCtxDN">ou=Customers,dc=example,dc=com</module-option>
            <module-option name="baseFilter">(uid={0})</module-option>
            <module-option name="rolesCtxDN">ou=Roles,dc=example,dc=com</module-option>
            <module-option name="roleFilter">(uniqueMember={1})</module-option>
            <module-option name="roleAttributeID">cn</module-option>
        </login-module>
    </authentication>
</application-policy>
```

To define security users/roles refer to your LDAP Vendors documentation. For example, if you use OpenLDAP, then with the ldif file customer-security.ldif, execute

```
ldapadd -x -D "cn=Manager,dc=example,dc=com" -w redhat -f customer-security.ldif
```

to setup users/roles.

Tip	module-options setting like url, bindDN, bindCredential, baseCtxDN, rolesCtxDN should match to your LDAP server setting.

To start Embedded Server with LdapExtLoginModule based security authentication via:

```
EmbeddedServer server =
...
EmbeddedConfiguration config = new EmbeddedConfiguration();
config.setSecurityDomain("teiid-security-ldap");
config.setSecurityHelper(new EmbeddedSecurityHelper());
server.start(config);
```

Teiid reference

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

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

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

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

Teiid 14.0.0 Release Notes

Teiid 14.0.0 adds performance features, microservice enablement, and fixes.

Release Notes

Highlights

- TEIID-5859 Upgraded to WildFly 19.1.0 including bringing the default standalone-teiid configuration up to date.
- TEIID-5927 TEIID-5935 The amazon-s3 translator is now capable of utilizing other s3 services including minio and ceph.
- TEIID-5917 The teiid.io website has been updated to better reflect the state of teiid related projects.

Compatibility Issues

- Support for named parameter syntax using param=value has been deprecated, since it is ambiguous with a comparison predicate boolean value expression. param⇒value should be used instead.
- decodeinteger/decodestring have been deprecated. A CASE expression should be used instead.
- TEIID-3159 The SAP Netweaver Gateway translator (sap-nw-gateway) has been renamed to just SAP Gateway (sap-gateway). Usage of sap-nw-gateway is deprecated.
- TEIID-4205 By default, the wrapping begin/commit of a UseDeclareFetch cursor will be ignored as Teiid does not require a transaction. Set the org.teiid.honorDeclareFetchTxn system property to false to revert to the old behavior which honored the transaction.
- TEIID-4240 The usage of ; delimited statements for materialization scripts has been deprecated. An anonymous procedure block should be used instead if multiple statements are needed.
- TEIID-4228 Precision and scale values greater than 32767 are deprecated.
- TEIID-4228 Not using a semicolon delimiter between statements is deprecated and should only be relied on for backwards compatibility.
- TEIID-4731 The default authentication scheme for MongoDB resource adapter has been changed to SCRAM_SHA_1 to match with latest versions of MongoDB. If using MongoDB version less than 3.0, to restore previous functionality change/add SecurityType to 'None' in the resource adapter configuration.
- TEIID-5511 The AdminShell kit has been removed. No further releases are planned.
- TEIID-5833 The use of SET NAMESPACE to define a custom prefix or namespace is no longer allowed. Please remove this statement from your DDL and simply use a consistent property key.

from 13.1

- TEIID-5948 The mysql5 translator name has been deprecated. You should use just mysql instead - the version will be detected from the source, or you may manually set the database version property.

from 13.0

- TEIID-5798 specifying a condition on a table permission is now deprecated. Use CREATE POLICY instead.

from 12.3

- TEIID-5819 References to any Teiid *-swagger artifact should use openapi in both the artifact name and group instead.
- TEIID-1323 The protected method SQLConverstion.generateSqlForStoredProcedure is now expected to append directly to the working buffer.
- TEIID-5565 The Teiid Java client now requires Java 8 and above.
- TEIID-5557 The default for the JDBC importer useFullSchemaName is now false. It is generally expected to import from only a single foreign schema. Set useFullSchemaName to true to preserve the legacy behavior. There is also an env/system property org.teiid.translator.jdbc.useFullSchemaNameDefault that can be used to preserve the legacy behavior.
- TEIID-5840 TEIID-5841 Grant / revoke targets are now resolved at deployment time and will be checked in a more strict manner. See the migration guide and/or the issues for more. The PolicyDecider was changed to reference the metadata objects rather than just strings.
- TEIID-5849 The admin and api modules/jars have been merged. Only teiid-api should be used moving forward.

- TEIID-5857 The salesforce translators no longer support the ModelAuditFields execution property - the import property should be used instead.

from 12.2

- TEIID-5742 The SecurityHelper interface has been simplified, instead of getSecurityContext() and getSubjectInContext(String), there is now just getSecurityContext(String)
- TEIID-5759 GRANT CONDITION syntax behavior did not match with XML VDBs and was updated to match - you must now explicitly use NOT CONSTRAINT to declare that the condition is not a constraint. If you still want the older behavior set the property org.teiid.conditionConstraintDefault to false.
- TEIID-5759 The odata4 openapi.json metadata url now returns v2 metadata by default. Please use /openapi.json?version=3 to get the v3 metadata.
- TEIID-5757 The teiid-security security domain is configured by default to provide the odata role, so that it does not have to be explicitly granted for odata access. If you wish to keep that requirement, then remove the Identity login module from the teiid-security security domain.
- TEIID-5729 The mapping of some procedures to OData functions will require explicitly setting the UPDATECOUNT option - CREATE VIRTUAL PROCEDURE … OPTIONS (UPDATECOUNT 0) AS BEGIN …

from 12.0

- TEIID-5640 Access to system schema over OData has been disabled. If you need access to SYS, SYSADMIN, or pg_catalog, consider adding an appropriate view or procedure.
- TEIID-5647 The information_schema schema name is now reserved for future internal use. If you do need to use this name for now, you can set the property org.teiid.allow_information_schema=true

from 11.2

- TEIID-5476 JGroups was removed as a direct dependency of the runtime and the associated property removed from the EmbeddedConfiguration. If you need clustering support with embedded, please raise an issue.
- TEIID-5563 All wildfly specific maven subprojects - including the resource adapter connector-x artifacts - were moved under the org.teiid.wildfly group id. See the Admin Guide for more migration information.
- TEIID-5596 The usage of infinispan caching with Teiid Embedded now requires a dependency to org.teiid:cache-infinispan.

from 11.1

- TEIID-5506 Removed the option to specify domain qualified logins.

from 11.0

- TEIID-5411 Pluggable server discovery has been removed as a client feature. The client will focus on better integration with existing load-balancing paradigms instead.
- TEIID-5415 The JDBC client load-balancing feature has been removed. The client will no longer pool instances nor issue a ping. If you use the client against a server older than 10.2, ping will need to be disabled on that server.
- TEIID-5427 Session/user scoping of materialized views has been removed. You should use a global temporary table instead and load it as needed for your session.

from 10.3

- TEIID-5365 Function model support has been completely removed from the server. VDBs utilizing function models should be migrated to having those functions located on physical or virtual models.
- TEIID-5083 The salesforce translator and resource adapter now provide 34.0 api access rather than 22.0.
- TEIID-5370 A warning rather than an exception will be generated when the HEADER option is specified for a TEXTTABLE, but the header/column does not exist in the file.
- TEIID-5360 JDBC DatabaseMetaData will no longer by default report nullsAreSortedLow as true since that behavior in not guaranteed and can be adjusted on the server side. If you need a particular value reported, use the connection property nullsAreSorted=\{AtStart,AtEnd,High,Low}

from 10.2

- TEIID-5294 The name escaping performed by the SQL/XML logic and JSONTOXML function did not properly escape values. Instead of *uHHHH*, *xHHHH* should have been used. That correction has been made. If you want the old behavior set the system property org.teiid.useXMLxEscape to false.

from 10.1

- TEIID-5286 The Sybase IQ translator has been renamed sap-iq and the usage of the SybaseIQExecutionFactory and the sybaseiq translator name has been deprecated.
- TEIID-5262 Removed support for Teiid 7.x clients/servers
- TEIID-5220 The pg_catalog now has information_schema.tables, views, and columns, which require qualification to reference the tables, views, or columns system tables.

from 10.0

- TEIID-5177 Stricter naming is now enforced in DDL. Only unqualified identifiers are expected as names. Set the system property org.teiid.requireUnqualifiedNames=false to restore the older behavior.
- TEIID-5201 The SYS.Keys table had SchemaUID and RefSchemaUID columns added.

from 9.x

- TEIID-4894 The XML document model feature has been removed. You must use OData or SQL/XML to create XML documents.
- TEIID-4924 Maven coordinates for Teiid artifacts have changed. They will now be pushed directly to Maven Central and will use the org.teiid group instead of org.jboss.teiid.
- TEIID-5026 The FROM_UNIXTIME function now returns a string rather than a timestamp value and no longer is rewritten to the timestampadd function. The functionality now matches that of HIVE/IMPALA. See also the to_millis and from_millis functions.
- TEIID-5012 A Description column was added to SYS.VirtualDatabases.
- TEIID-4943 Copy criteria created from a join will typically only be pushed when the join is not pushed.
- TEIID-5112 Type length specified in DDL or SQL must be greater than 0. Char type length must only be 1.
- TEIID-5130 Procedure RESULT parameters must appear as the first parameter in the argument list. To allow the old behavior of appearing anywhere, set the system property org.teiid.resultAnyPosition=true.

- TEIID-3624 The introduction of domain types modified several of the system tables. The isPhysical column was removed from the SYS.Datatypes table. SYS.Datatypes added Type, TypeCode, Literal_Prefix, and Literal_Suffix columns. The SYS.Columns, SYS.ProcedureParams, and SYS.FunctionParams tables added TypeName, TypeCode, and ColumnSize columns.
- TEIID-4827 Java 1.8 is now required for building and running Teiid.
- TEIID-4890 The ProcedureParameters system table will report return parameters as position 0.
- TEIID-4866 For usability with SQLAlchemy and Superset the version() function over ODBC will report ""PostgreSQL 8.2" rather than "Teiid version". You can use the system property org.teiid.pgVersion to control this further.
- TEIID-4574 Phoenix/Hbase Translator has been renamed phoenix and the usage of the HBaseExecutionFactory and the hbase translator name has been deprecated.
- TEIID-4501 The salesforce-34 resource adapter defaults to the version 34 api rather than version 22 api.
- TEIID-3754 OData Version 2 support is removed. Please use OData V4. Note that there are many changes in specification with V4 vs V2.
- TEIID-4400 XML Document Models have been deprecated. OData or SQL/XML should be used instead.
- TEIID-4317 ExecutionFactory.initCapabilities will always be called - either during start if isSourceRequiredForCapabilities returns false, or later if true.
- TEIID-4346 The excel-odbc translator has been removed. Please use the excel translator instead.
- TEIID-4332 Due to costing logic changes plans may be different that in previous releases. Please raise an issue is you feel a plan is not appropriate.
- TEIID-4421 Removed the deprecated EmbeddedServer.addTranslator(ExecutionFactory) method.
- TEIID-4442 Removed the interpretation of the security-domain setting for the session service as a comma separated list of domains. Also added the USER(boolean) function to control if the USER function returns a name with the security domain. Finally the DatabaseMetaData and CommandContext getUserName will both return the simple user name without the domain.
- TEIID-4228 Precision/scale will now be set consistently. Values reported from JDBC/OData/ODBC metadata may be different if your current metadata declares a bigdecimal type with default precision.
- TEIID-4423 Uncorrelated subqueries will be treated as deterministic regardless of functions used within them. Prior releases treated most uncorrelated subqueries as non-deterministic if they contained a non-deterministic function.

from 8.x

- TEIID-2694 In the autogenerated web service, if a procedure is designed for POST method, and one of its IN/INOUT parameters is either a LOB or VARBINARY then that service can only invoked using "multipart/form-data". This allows user to send large binary files for processing in Teiid
- TEIID-3462 Semantic versioning requires the VDB version to be a string, rather than an integer field. This affects several public classes including CommandLogMessage, VDB, Session, EventListener, VDBImport, ExecutionContext, and MetadataRepository. Any custom command logging or materialization status tables will need the version field updated as well.
- TEIID-4147 ODBC type handling will now report the type name as the PostgreSQL type rather than the Teiid type.
- TEIID-3601 changed the rowCount field on CommandLogMessages from Integer to Long.
- TEIID-3752 the admin assignToModel method was removed

- TEIID-3684 RoleBasedCredentialMapIdentityLoginModule removed, consider using alternative login modules with roles to restrict access to VDB
- TEIID-2476 The AuthorizationValidator and PolicyDecider interfaces had minor changes - see their javadocs for new/altered methods
- TEIID-3503 To better isolate dependencies a separate teiid-jboss-admin jar was created from classes in teiid-admin - most notably AdminFactory was moved there.
- TEIID-4206 TranslatorProperty annotations on methods without setters must have the readOnly attribute as true.
- TEIID-3814 In the autogenerated web service, the model name in the path is now case sensitive.
- TEIID-2267 The custom appenders for command and audit logging has been changed, now they need to be developed for java.util.logging based Handler.
- TEIID-3553 Ambiguous OData v2 entity set and function names will throw an exception rather than resolving to the first found.
- TEIID-3515 MAKEIND was added as a reserved word.
- TEIID-3576 the waitForLoad connection property has been deprecated.
- TEIID-2813 a source end event will be sent to the command log when an error occurs rather than being omitted.
- TEIID-3736 string literals values matching the date format can be directly resolved as timestamps.
- TEIID-3727 The version 22 salesforce translator and resource adapter have been deprecated.
- TEIID-3380/https://issues.redhat.com/browse/TEIID-3663[TEIID-3663] The SecurityHelper interface has changed to allow for easier control over GSS authentication
- TEIID-3372 DDL and DDL-FILE metadata repositories have deprecating using the respective ddl and ddl-file model properties.
- TEIID-3390 temporary lobs are now cleaned up when the result set is closed - even for local connections.
- TEIID-3210 Added supportsCompareCriteriaOrderedExclusive, which defaults to supportsCompareCriteriaOrdered, to specifically support < and > pushdown.
- TEIID-3282 Changed the WEEK function to compute the ISO 8601 by default (org.teiid.iso8601Week=true) and ensured pushdowns do the same. Changed the dayOfWeek function to be unaffected by the iso8601Week setting.
- TEIID-2904 The createMetadataProcessor method on JDBCExcutionFactory has been deprecated. Use getMetadataProcessor instead.
- TEIID-2793 Searchability metadata will not prevent more complicated expressions from being pushed down.
- TEIID-2794 Schema scoped functions are checked for ambiguity. Schema qualification may be needed to resolve properly.
- TEIID-2840 Internal materialized view ttl refresh is now blocking by default. To keep the old behavior of lazy invalidation, use the vdb property lazy-invalidation=true
- TEIID-2667 The jdbc importer importKeys parameter is now correctly defaulted to true.
- TEIID-2737 The 'native' procedure exposed by translators has been renames as the direct query feature. The related ExecutionFactory methods supportsNativeQueries and nativeQueryProcedure name have been deprecated and replaced with supportsDirectQueryProcedure and directQueryProcedureName.
- TEIID-2580 Both xpathValue and XMLTABLE will return null when retrieving the value for a single element marked with xis:nil="true".

- TEIID-2590 Both the source specific and the general hint if present will be included as the source hint for Oracle.
- TEIID-2603 TableStats and ColumnStats numeric values are held as Number, rather than Integer.
- TEIID-2613 The rowcount is reset to 0 after a non-update command statement is issued.
- TEIID-2422 using calendar based timestampdiff by default. See the Admin Guide for using the org.teiid.calendarTimestampDiff to control backwards compatibility.
- TEIID-2477 Most of the JDBC translator static String version constants were replaced by org.teiid.translator.jdbc.Version constants. Use the .toString() method to obtain a version string if needed.
- TEIID-2344 non-available JDBC sources in partial results mode or source with connection factories that require an ExecutionContext to obtain a connection will require manual setting of the database version metadata property. The affected sources are: db2, derby, oracle, postgresql, sqlserver, sybase, teiid
- TEIID-2444 The deployment platform for Teiid has been changed to EAP 6.1.Alpha1, older or non-EAP deployments are not supported.
- TEIID-2429 Sorts over data sets over a single batch are not guaranteed to be sorted in a stable manor to improve performance. The sort will still be correct with respect to the sort keys.
- TEIID-1979 The resource adaptors are now deployed through modules, and have shorter names as identifiers. Connection Factories created with previous versions must be re-configured.
- TEIID-2253 the multi-source implementation logic was significantly altered the following changes were introduced.
  - If not auto-populated, the multi-source column acts as a pseudo-column and will not be selectable via a wildcard SELECT * nor tbl.*
  - Multi-source inserts must specify a single source as their target.
  - The join planning behavior in multi-source mode was not consistent and did not work in all situations. To ensure consistency multi-source tables being joined together should specify a join predicate on the source name column - i.e. tbl1.source_name = tbl2.source_name. For backwards compatibility a the system property org.teiid.implicitMultiSourceJoin was introduced to control whether multi-source joins are effectively partitioned by source without a source_name predicate. The property defaults to true, the pre 8.3 behavior - but should be switched to false for later versions unless the issues with implicit join planning are addressed.
- TEIID-2317 byte[] char[] and java.util.Date instances returned as object values will be left in tact and not automatically converted to BinaryType, ClobType, and Timestamp respectively. The values may still be cast to those types.
- TEIID-2149 the subqueryUnnestDefault property no longer influences cost based decisions to treat subqueries as merge joins. In nearly all circumstances this is desirable, but may require the use of nounnest hint to prevent forming the join if desired.
- TEIID-2166 array_get will return null if the index is out of bounds rather than raising an error.
- TEIID-2175 for 8.0 and 8.1 clients the server will check if serialized date/time values fall outside of 32-bit value ranges (year 1900 - 9999 for dates and times between years 1901 and 2038) and throw an exception. The previous behavior was to truncate. The exception and the use of 32 bit serialization can be avoided by setting the system property org.teiid.longDatesTimes to true.
- TEIID-2184 to be consistent with the rest of Teiid’s logic the system functions dayName and monthName will return values from the default locale, rather than only the English names. Use the system property org.teiid.enDateNames true to revert to the pre-8.2 behavior.
- TEIID-2187 the CONSTRAINT keyword is not correctly used in table DDL. It should be replaced with a comma from scripts to be compatible with 8.2. If desired, 8.2 now supports the CONSTRAINT keyword to provide a name for each constraint.

- TEIID-2181 system tables no longer contain valid OIDs. That responsibility has moved to the pg_catalog.
- TEIID-1386 the SQLState and errorCode reported by a TeiidSQLException will typically be from the top level nested SQLException. If there is also a nested TeiidException, the TeiidSQLException.teiidCode will be set to the TeiidException.getCode value and the TeiidSQLException.errorCode will be set to the integer suffix of the teiidCode if possible.
- TEIID-2226 All statements that return result sets that are executed as command statements in a procedure are validated against the expected resultset columns of the procedure. If the statement is not intended to be returnable, WITHOUT RETURN can be added to the end of the statement.
- TEIID-2235 The MetadataRepository.setNext method was removed and MetadataRepository was converted to an abstract class rather than an interface. Also if an instance of a DefaultMetadataRepository is used, it will only affect metadata already loaded in the repository chain.
- TEIID-2237 teiid_ is a reserved DDL namespace prefix and the MetadataFactory class no longer throws TranslatorExceptions, instead the unchecked MetadataException is thrown.
- TEIID-2243 by default Teiid will not pushdown the default null sort order of nulls low when no null sort order is specified. Set the system property org.teiid.pushdownDefaultNullOrder to true mimic the 8.1 and older release behavior.
- org.teiid.metadata.Schema holds FunctionMethods by uuid rather than name to accommodate overridden method signatures.
- MetadataFactory no longer extends Schema. Use the MetadataFactory.getSchema method to get the target Schema.
- DDL created VIRTUAL pushdown functions should be referenced in the ExecutionFactory.getSupportedFunctions by their full schema.function name.
- DDL functions/procedures defined without the VIRTUAL keyword are by default VIRTUAL. Use the FOREIGN keyword to indicate that they are source specific.
- FunctionMethod.getFullName returns the proper schema, not category qualified name.
- VDB.getUrl has been removed.
- VDB.Status now has four states - LOADING, ACTIVE, FAILED, REMOVED. To check for validity use the isValid method, rather than checking for the VALID state. FAILED deployments will still be accessible via the admin getVDB methods.
- The standalone and cli configuration files specify a setting for the teiid subsystem policy-decider-module. If a module is not specified, then data roles will not be checked.
- local connections specifying a VDB version will wait for their VDB to finish loading before allowing a connection, see the waitForLoad connection property for more.
- jsonToXml document elements will contain xsi:type attribute values of decimal and boolean respectively for number and boolean json values to allow for differentiation from string values.
- Result set cache entries can now have updatable set to false to indicate that updates should not purge the entry.
- Datatype default values have been corrected for Teiid built-in types. All datatypes are now nullable by default, only character string types are case sensitive, numeric types have radix 10, and length/precision/scale have been set appropriately.
- pg catalog and dynamic vdb created metadata will use a generated Teiid id rather than a random UUID.
- transport ssl config no longer uses the enabled attribute. Use mode=disabled to disable the usage of encryption.
- TEIID-2105 If a MetadataRepository throws a RuntimeException during load, that will be treated as a non-recoverable error and the VDB will have a FAILED status.
- TEIID-2105 It was an undocumented behavior that is a source did not specify a jndi connection that "java:/name" would be assumed. That is no longer the case. It the source needs a connection, then one must be specified.

- TEIID-2127 if ExecutionFactory.isSourceRequired returns true (the default) then not obtaining a connection will for an Execution will result in an error. If an ExecutionFactory does not use a source, then no connection-jndi-name should be specified and isSourceRequired should return false (see setSourceRequired). If isSourceRequired returns false and a connection-jndi-name is specified, then Teiid will still attempt to obtain a connection, but no exception will be thrown if a connection isn't available.
- TEIID-2138 the odbc layer will report standard_conforming_strings as on, rather than off to better reflect the string literal handling of Teiid.

Configuration Issues

See the Admin Guide for more on configuration and installation.

from 12.0

- TEIID-5642 The generic sql query procedure for generated REST wars will not be exposed by default. The schema/model must have the property \{http://teiid.org/rest}sqlquery set to true.

from 11.2

- TEIID-5584 org.teiid.enforceSingleMaxBufferSizeEstimate now defaults to false. Rather the biggest memory consumers among sessions will be killed by default in the event of running out of disk space.
- TEIID-5490 org.teiid.longRanks now defaults to true. Analytical functions such as row_number return a long by default.
- TEIID-5574 the cli buffer-service properties have been deprecated and replaced with buffer-manager properties - see the migration guide for more

from 10.3

- TEIIDTOOLS-381 the default max buffer space for Teiid embedded and derived runtimes (Thorntail/Spring Boot) is 5 gigabytes, rather than 50. For the full WildFly environment the default is still 50 gigabytes (51200 megabytes), via the stanadlone-teiid buffer-service max-buffer-space attribute.

from 10.2

- TEIID-5323 User query command log entries are now logged at the DEBUG level on the org.teiid.COMMAND_LOG context. Source events are logged on the org.teiid.COMMAND_LOG.SOURCE context at the DEBUG level. This allows command logging of just the user query events by setting the logging level to DEBUG for the overall context, but INFO or higher for the SOURCE child context. The level will default to WARN in the standard install or to DEBUG when running the auditcommand scripts.

from 10.1

- TEIID-5248 v4 Api Support modified the properties for the Google Resource Adapter. The Key property was removed - use SpreadsheetId instead. The AuthMethod property was removed as well.
- TEIID-5268 Anonymous authentication requires setting the LdapAuthType property to none on the LDAP Resource Adapter.

from 9.x

- TEIID-4820 The JDG specific connectivity is being separated from the main community project. It will be made available separately and as part of the product.

- TEIID-4858 The Hive translator now has order by support turned off by default.
- TEIID-4533 The default for the max-staleness of the resultset cache was changed from 60 seconds to 0 seconds. You may use the cli to alter this new default if necessary.
- TEIID-4707 The PrestoDB driver is no longer pre-installed. This allows for newer client versions to be used as needed. The documentation has been updated to reflect this as well.
- TEIID-4129 in order to prevent invalid results from a sort/merge join, the sort operation will undergo additional checks. If org.teiid.assumeMatchingCollation is false (the default) and a translator does not specify a collationLocale, then the sort for a sort/merge join will not be pushed. Teiid defaults to the Java UCS-2 collation, which may not match the default collation for sources, particular tables, or columns. You may set the system property org.teiid.assumeMatchingCollation true to restore the old default behavior or selectively update the translators to report a collationLocale matching org.teiid.collationLocale (UCS-2 if unset).

from 8.x

- TEIID-2754 view are reported as VIEW table type in the metadata. Use the connection property reportAsViews=false to restore the old behavior.
- TEIID-3753 org.teiid.widenComparisonToString now defaults to false.
- TEIID-3669 there is now a single session service. Common configuration properties need to be consolidated. With TEIID-3790 this also means that you may want to change the default of trust-all-local to false to restrict local pass-through connections. Also the VDB REST passthrough-auth property is no longer used.
- TEIID-3797 the embedded transport is now known as the local transport.
- TEIID-3859 the "native" 9999 management port is no longer used. AdminShell will default to the http 9990 management port instead.
- TEIID-3594 User query command log entries are now logged at the INFO level on the org.teiid.COMMAND_LOG context. This allows command logging of just the user query events by setting the logging level to INFO. The level will default to WARN in the standard install or to DEBUG when running the auditcommand scripts.
- TEIID-3192 The CXF config is no longer a valid option for the Salesforce resource adapter. Please log an issue if there is feature from the CXF config that you were using that is not present on the new resource adapter.
- TEIID-3177 ODBC connections will be required to be secure based upon the SSL mode setting. If the mode is enabled, then the client must request an SSL connection. If the mode is login, then the client must use GSS authentication. To revert to the prior behavior, the system property org.teiid.ODBCRequireSecure can be set to false.
- TEIID-2512 the usage of the metadata element text as the "raw schema text" may not be appropriate in all situations. The ddl and ddl-file metadata repositories will check for the ddl and ddl-file model properties respectively.
- TEIID-2707 the org.teiid.joinPrefetchBatches property is no longer used.
- TEIID-2429 the default for maxProcessingKb has effectively doubled (the old default would use approaximately 4MB), while the maxReserveKb default has been reduced to 70% of the memory past the first gigabyte instead of 75%.
- TEIID-2445 the UseConnectorMetadata and supports-multi-source-bindings properties have been deprecated, but will still be respected if present. There is no equavalent to UserConnectorMetadata=true as it is always implied. UseConnectorMetadata=false has been replaced by cache-metadata=false, which can be placed at either the vdb or model level. supports-multi-source-bindings has been replaced by multisource, which no longer needs to be specified if more than one source is configured.
- TEIID-2510 the time-slice-in-millseconds has been corrected to be time-slice-in-milliseconds

- The connector batch size setting is no longer used. Instead a fetch size will be sent to the translator that is 2 times the working batch size or the non-pushed limit, whichever is less.
- The file translator now defaults to exceptionIfFileNotFound=true, you can set the translator property to false to preserve the old behavior of returning null.
- TEIID-2086 TEIID-2168 prepared plan and result set caches are now configured as infinispan caches. See the teiid cache container in the configuration. You may also control the transactional aspects of the result set cache on the resultset and resultset-repl caches via the configuration.
- TEIID-1241 the web services connector property ConfigName was deprecated in favor of EndPointName. There were also ServiceName, NamespaceUri, and Wsdl properties added, which are used to point the
- teiid-security-users and teiid-security-roles properties files have been moved under the configuration directory of their respective deployment.

Other Issues

- TEIID-5687 - Querying NCHAR values in Oracle using prepared statements and unicode values will result in the value being converted to extended ascii instead.
- TEIID-1281 - Negative start indexing is not supported by DB2 and Derby databases. Usage of the Teiid SUBSTRING against these sources should not use negative start values.
- TEIID-1008 - Most versions of Oracle and MySQL do not support deeply nested correlated references. There is currently no workaround for this issue.
- For compatibility with the 7.0 release if a stored procedure parameter list begins with identifier=, then it will be parsed as a named parameter invocation even if the intent was to use a comparison predicate as the first parameter value. The workaround is to use nesting parens, e.g. call proc((identifier=value), …), which clarifies that this is positional value. This workaround will not be needed in later releases.
- TEIID-586 - Salesforce LIKE pushdown is case insensitive, while LIKE evaluated by Teiid is case sensitive unless an alternative collation is used. Care should be taken to ensure consistent results if mixed case values are being searched.
- TEIID-2836 - Data from DB2 on z/OS in EBCDIC may not be represented correctly at runtime. It is recommended that the values are converted to ASCII or another common character set.
- TEIID-2998 - Google spreadsheets containing all string data do not detect their row data and labels correctly on the Google backend.
- TEIID-3070 - Netty threads may inappropriately take up CPU resources. This affects most EAP releases. Upgrade the AS version of Netty to 3.6.10.Final to address this issue.
- TEIID-3289 - The timestamp to string conversion performed in MySQL will produce a string with all of the trailing zeros (up to 6) for the fractional seconds. This differs from the expected Teiid/Java format.
- TEIID-2836 - Data from DB2 on z/OS in EBCDIC may not be represented correctly at runtime. It is recommended that the values are converted to ASCII or another common character set.
- TEIID-2998 - Google spreadsheets containing all string data do not detect their row data and labels correctly on the Google backend.
- TEIID-3070 - Netty threads may inappropriately take up CPU resources. This affects most EAP releases. Upgrade the AS version of Netty to 3.6.10.Final to address this issue.
- TEIID-3289 - The timestamp to string conversion performed in MySQL will produce a string with all of the trailing zeros (up to 6) for the fractional seconds. This differs from the expected Teiid/Java format.

- TEIID-3779 - There are a host of Phoenix issues that Teiid is currently not working around for HBase access. If you hit any of these, please let us know so that we can work with the Phoenix community to get it resolved. Generally Phoenix has issues with subquery evaluation and certain datatypes, such as char and timestamp.
    - TEIID-3772 TEIID-3769 TEIID-3766 are not likely to occur and generate an exception.
    - TEIID-3774 is unlikely but can return inaccurate results.
    - TEIID-3768 affects correlated subquery comparison using an aggregate of a char value and can return inaccurate results.
- 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.
- TEIID-3805 - SAP Hana returns an empty string rather than null for the substring function when the from index is larger than the string length.
- TEIID-3816 - Informix can return incorrect results for subquery comparisons involving a boolean value and a subquery that has only a single row. If you encounter such a scenario and need Teiid to compensate, then please open an issue.

Thirdparty Library Updates

The following components have been updated:

From 12.3

- The infinispan-hotrod translator/resource adapter were updated to Infinispan 10.0.1.
- Olingo was upgraded to 4.7

From 12.1

- The salesforce-41 translator/resource adapter were updated to the 45.1.0 jars.
- Olingo was upgraded to 4.6

From 12.0

- Apache POI for the excel translator was upgraded to 3.13.
- Accumulo core and related dependencies were updated to 1.9.2.
- The mongodb driver was upgraded to 3.9.1.
- jts and related were updated to 1.16.0

From 11.0

- The cassandra driver and associated dependencies were upgraded to 3.5.1.

From 10.1

- Saxon was upgraded to 9.8.0-7.
- The MongoDB client was upgraded to 3.6.3

From 10.0

- The Swagger libraries were updated to version 1.5.17, and the swagger-parser was upgraded to version 1.0.33.

Detailed Release Notes

Detailed Release Notes - Teiid - Version 14.0.0

Documentation and Help

The Teiid community project is hosted on jboss.org. Documentation and help may be obtained from the local distribution under teiid-docs or the following locations.

- Online Documentation
- Wiki
- JIRA
- Forums

Licenses

Teiid is primarily licensed under the Apache Software License 2.0. Individual jars built for Teiid are also licensed under the EPL, MPL, and the PostgreSQL-BSD licenses as per the needs of their originating source. See the license directory in the distribution for full license copies. Third-party jars retain their original licensing.

About Red Hat

Red Hat, is in the business of providing superior technical support to our customers. Our goal is to make Professional Open Source™ the **SAFE CHOICE** for you. We accomplish this by backing up our open source Java products with technical support services that are delivered by the core developers themselves. We can help you to train your staff and provide you with support at every stage of the application lifecycle - from development and integration through deployment and maintenance. Visit the JBoss Services page for more information.

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
- OpenAPI / Swagger
- 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

OpenAPI / Swagger

See the OpenAPI and Swagger Translators

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 and file sources (file and ftp) for use with:

- Delimited/Fixed width
- XML
- JSON

Spreadsheets

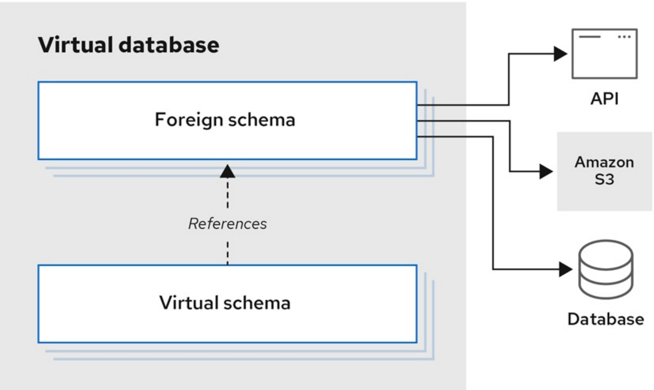
- Excel
- Google Spreadsheet

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



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

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

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

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

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

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

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

Or as an XML vdb:

```
<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>
```

Both formats define the same VDB.

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

External data sources

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

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

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

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 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
    VERSION 'one' FOREIGN DATA WRAPPER postgresql
    OPTIONS (
        "resource-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.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:r_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 the SQL-MED specification to utilize foreign data sources.

DDL captures information about the VDB - the sources it integrate, and preferences for importing metadata. DDL may be deployed as a single file or as a set of files in a zip archive.

See Developing a Virtual Database for a discussion of the .vdb zip packaging.

Table of Contents

DDL File Deployment

You can simply create a **SOME-NAME-vdb.ddl** file with your DDL content. Then use a standard deployment mechanism (cli, adminapi, or placing the file in the deployments directory) to deploy it.

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

```
CREATE DATABASE my_example;
USE DATABASE my_example;

CREATE SERVER pgsql
    VERSION 'one' FOREIGN DATA WRAPPER postgresql
    OPTIONS (
        "resource-name" 'java:/postgres-ds'
    );

CREATE SCHEMA test SERVER pgsql;
IMPORT FOREIGN SCHEMA public FROM SERVER pgsql INTO test
    OPTIONS(
        importer.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 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 the list of 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 translators include:

- oracle
- mysql
- postgresql
- mongodb

```
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 an "override" translator. It is not required to define translators already known to the engine with a CREATE - for example CREATE FOREIGN DATA WRAPPER oracle OPTIONS … - will effectively be ignored.

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:as_translators.adoc[Translators] documents all the available translators.

For all available translators see Translators

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}
```

Associate The Translator With A Source

The SERVER construct is used to associate your translator with a data 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	Not currently used.
translator-name	Name of the translator to be used with this server.
options	Currently only resource-name is supported. resource-name provides a way to specify the environmentally dependent (JNDI or bean) name of the source if it differs from the server name. For example java:/source

Example 3: creating a data source connection to Postgres database

```
CREATE SERVER pgsql
    FOREIGN DATA WRAPPER postgresql
    OPTIONS (
        "resource-name" 'java:/postgres-ds'
    );
```

An example file source.

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

```
CREATE SERVER marketdata
    FOREIGN DATA WRAPPER file
    OPTIONS(
        ParentDirectory '/path/to/marketdata' ,"resource-name" 'java:/postgres-ds'
    );
```

See Data Sources for more.

Additional management support to alter/delete a connection.

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

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

Create SCHEMA

A schema is a container for metadata. It 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.

Below are typical properties that can be configured for a Schema in the OPTIONS clause.

Name	Description
VISIBLE	Set to false to make the Schema not visible to metadata interrogation
ANNOTATION	A description of the Schema

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 to import. Typically most databases are tied to a schema name, like "public", "dbo" or name of the database. If you are working with a non-relational source, or a DDL file, you can provide a dummy value here or omit the entire FOREIGN SCHEMA clause. 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 Import

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

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

Example Import

```
-- in archive based vdbs(.vdb) you can provide schema in separate files and pull
them in a main vdb.ddl file as:
IMPORT FROM REPOSITORY "DDL-FILE"
    INTO test OPTIONS ("ddl-file" '/path/to/schema1.ddl')
IMPORT FROM REPOSITORY "DDL-FILE"
    INTO test OPTIONS ("ddl-file" '/path/to/schema2.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.

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
    FOREIGN ROLE {enterprise-role}(,{enterprise-role})*
    | 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 FOREIGN 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} |
        PROCEDURE {schema-name}.{procedure-name} |
        SCHEMA {schema-name} |
        COLUMN {schema-name}.{table-name}.{column-name} [MASK [ORDER n] {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} |
        PROCEDURE {schema-name}.{procedure-name} |
        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
- expression: any valid sql expression, this can include columns from referenced resource

BNF for CREATE POLICY and DROP POLICY.

Warning	GRANT/REVOKE mostly function as direct replacements for the legacy permission model. They do not function the same as standard SQL GRANT/REVOKE. GRANT/REVOKE apply/remove permissions from the given resource - but do not affect prior GRANT/REVOKEs against any other resource. For example if you GRANT the select permission on a table, then REVOKE the select permission on the table's schema, the GRANT of the select permission will remain on the table. At runtime GRANTs are still interpreted hierarchically - a select GRANT on a schema implies read access to all contained schema objects. GRANT/REVOKE is also not ADD/DROP aware. If the GRANT target is dropped the old GRANT still remains and could affect any recreated object.

Warning	POLICIES are not ADD/DROP aware. If the POLICY target is dropped the old POLICY still remains and could affect any recreated object.

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

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

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

```
CREATE ROLE everything WITH FOREIGN ROLE admin;
```

```
...
GRANT ALL PRIVILEGES TO everything;
```

Example : All users can see only Orders table contents amount < 1000

```
CREATE ROLE base_role WITH ANY AUTHENTICATED;
...
GRANT SELECT ON TABLE test.Orders TO base_role;
CREATE POLICY policyOrders ON test.Orders TO base_role USING (amount < 1000) TO
base_role;
```

Example : Override previous example to more privileged user.

```
CREATE POLICY policyRoleAOrders ON test.Orders TO RoleA USING (amount < 1000 and amount >=1000);
```

Example : Restricting rows to only those owned by this user.

```
GRANT SELECT ON TABLE test.CustomerOrders TO RoleA;
CREATE POLICY policyCustomerOrders ON test.CustomerOrders TO RoleA USING (name = user());
```

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 SELECT ON COLUMN test.Order.amount
    MASK 'xxxx'
    TO base_role;
```

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

```
GRANT SELECT 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 SELECT 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 deployment 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 properties

DATABASE properties

- *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 virtual database. `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. For more information, see *Support for User-Defined Functions (Non-Pushdown)* in the Translator Development Guide.
- *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. For more information about related properties, see *Driver Connection* in the Client Developer’s Guide. Note these are set on the connection after it has been established.

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

```
<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 over *authentication-type*. The default is *authentication-type*.

- *gss-pattern*

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

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

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

- *model.visible*

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

- *include-pg-metadata*

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

- *lazy-invalidate*

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

- *deployment-name*

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

Schema and model properties

- *visible*

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

- *multisource*

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

- *multisource.columnName*

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

- *multisource.addColumn*

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

- *allowed-languages*

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

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

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

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

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

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

    </data-role>
```

DDL metadata for schema objects

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

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

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

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

Example: Create foreign table (Created on PHYSICAL model)

```
CREATE FOREIGN TABLE {table-name} (
    <table-element> (,<table-element>)*
    (,<constraint>)*
) [OPTIONS (<options-clause>)]


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

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

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

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

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

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

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

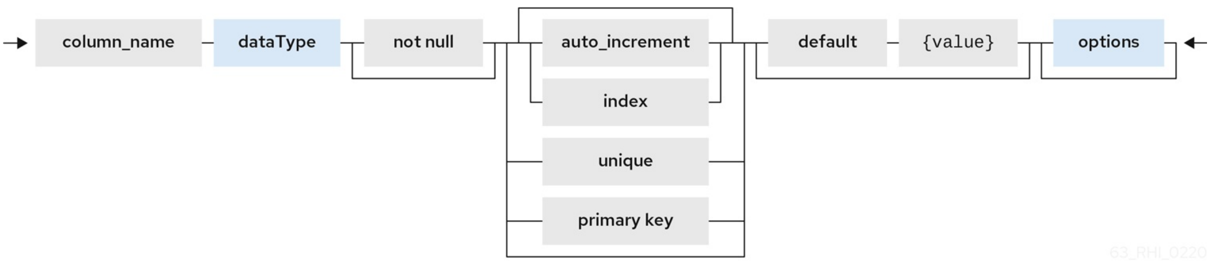
Example: Create foreign table (Created on PHYSICAL model)

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

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

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

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



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

Property	Data type or allowed values	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 type.
MIN_VALUE		
MAX_VALUE		
CHAR_OCTET_LENGTH	integer	
ANNOTATION	string	
NATIVE_TYPE	string	
RADIX	integer	
NULL_VALUE_COUNT	long	Costing informatior Number of NULLS in this columr

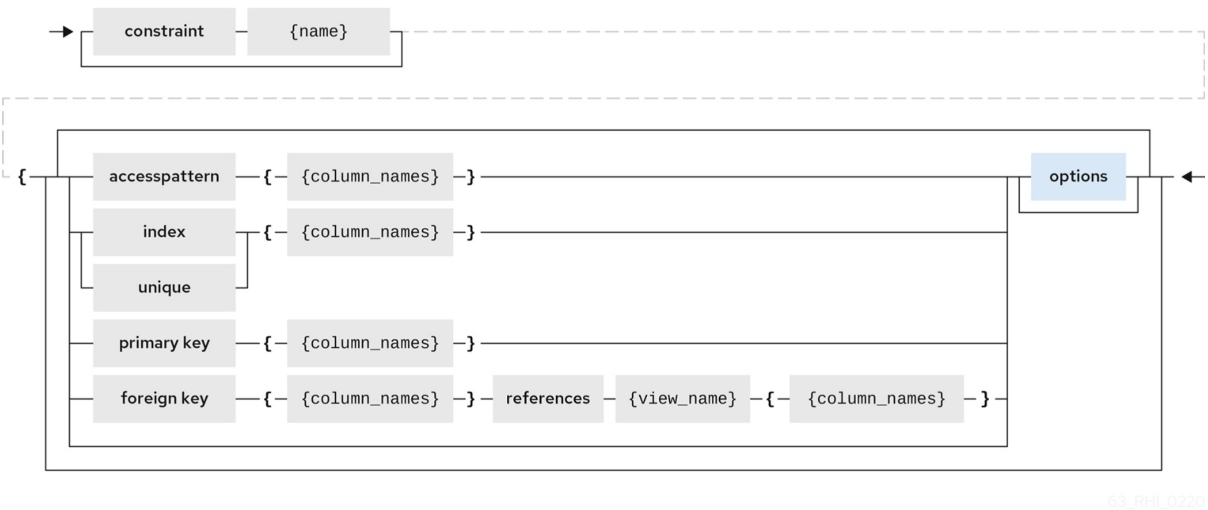
DISTINCT_VALUES	long	Costing informatior Number of distinct values in th column.

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

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

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

Table Constraints
Constraints can be defined on table/view to define indexes and relationships to other tables/views. This information is used by the 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
For the full SQL grammar for the ALTER TABLE statement, see *ALTER TABLE* in the BNF for SQL grammar.

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

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

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

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

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

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

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

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

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

Views

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

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

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

BNF for CREATE VIEW

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


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

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

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

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

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

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

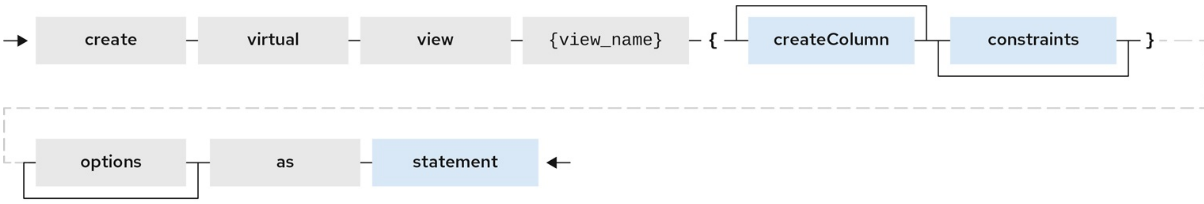


Table 1. **VIEW OPTIONS**: (These properties are in addition to properties defined in the CREATE TABLE)

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

Example: Create view table (created on VIRTUAL schema)

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

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

ALTER TABLE

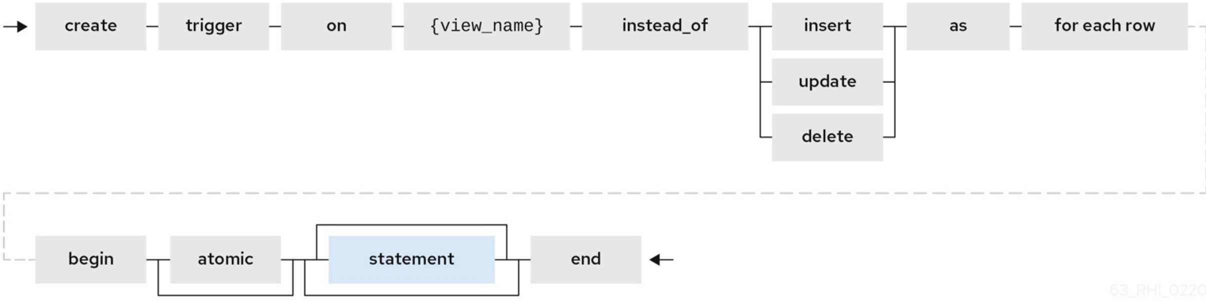
The BNF for ALTER VIEW, refer to ALTER TABLE

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

```
ALTER VIEW CustomerOrders
    AS
    SELECT concat(c.firstname, c.lastname) as name,
        o.saledate as saledate,
        o.amount as amount
  FROM Customer C JOIN Order o ON c.id = o.customerid
  WHERE saledate < TIMESTAMPADD(now(), -1, SQL_TSI_MONTH)
```

INSTEAD OF triggers on VIEW (Update VIEW)

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



Example: Define INSTEAD OF trigger on View for INSERT

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

For Update

Example: Define instead of trigger on View for UPDATE

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

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

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

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

Example:Define AFTER trigger on Customer

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

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

For more detailed information about update procedures, see Update procedures

Create procedure/function
A user can define one of the following functions:

Source Procedure ("CREATE FOREIGN PROCEDURE")

A stored procedure in source.

Source Function ("CREATE FOREIGN FUNCTION")

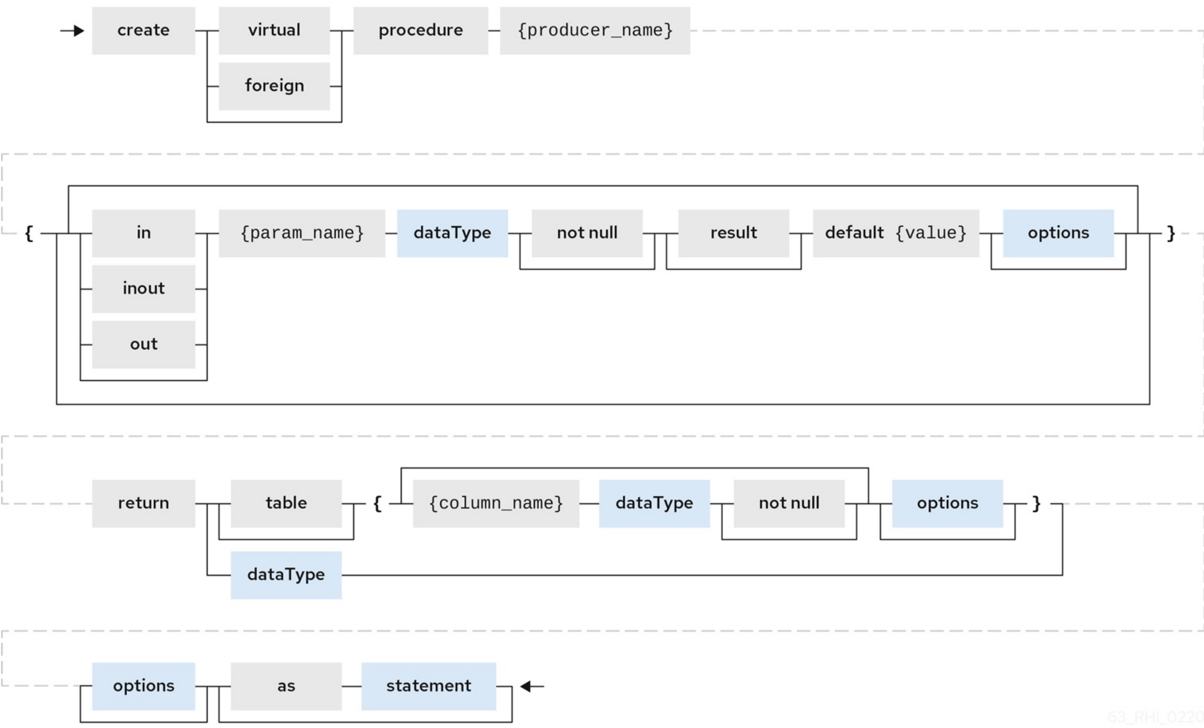
A function that depends on capabilities in the data source, and for which Teiid will pushdown to the source instead of evaluating in the Teiid engine.

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 than can have the implementation defined by a Java class. For more information about writing the Java code for a UDF, see *Support for user-defined functions (non-pushdown)* in the Translator Development Guide.



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

Variable arguments

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

Example: Vararg procedure

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

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

Property	Data Type or Allowed Values	Description
UUID	string	unique Identifier

NAMEINSOURCE	If this is source function/procedure the name in the physical source, if different from the logical name given above	
ANNOTATION	string	Description of the function/procedure
CATEGORY	string	Function Category
DETERMINISM	NONDETERMINISTIC, COMMAND_DETERMINISTIC, SESSION_DETERMINISTIC, USER_DETERMINISTIC, VDB_DETERMINISTIC, DETERMINISTIC	Not used on virtual procedures
NULL-ON-NULL	'TRUE'|'FALSE'	
JAVA_CLASS	string	Java Class that defines the method in case of UDF
JAVA_METHOD	string	The Java method name on the above defined java class for the UDF implementation
VARARGS	'TRUE'|'FALSE'	Indicates that the last argument of the function can be repeated 0 to any number of times. default false. It is more proper to use a VARIADIC parameter.
AGGREGATE	'TRUE'|'FALSE'	Indicates the function is a user defined aggregate function. Properties specific to aggregates are listed below.

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

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

.AGGREGATE FUNCTION OPTIONS

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

DECOMPOSABLE	'TRUE'|'FALSE'	Indicates the single argument aggregate function can be decomposed as agg(agg(x)) over subsets of data. The default value is `false` .
USES-DISTINCT-ROWS	'TRUE'|'FALSE'	Indicates the aggregate function effectively uses distinct rows rather than all rows. The default value is `false` .

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

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

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

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

Example: Define virtual procedure

```
CREATE VIRTUAL PROCEDURE CustomerActivity(customerid integer)
    RETURNS (name varchar(25), activitydate date, amount decimal)
    AS
    BEGIN
        ...
    END
```

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

Example: Define virtual function

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

Procedure columns may also be marked as NOT NULL, or with a DEFAULT value. On a source procedure if you want the parameter to be defaultable in the source procedure and not supply a default value in 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 parentheses. For more information, see *Parameterizable native queries* in Translators.
non-prepared	boolean	Applies to JDBC procedures using the native-query option. If true a PreparedStatement will not be used to execute the native query.

Example: Native query

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

Example:Sequence native query

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

Tip	Use source function representations to expose sequence functionality.

Extension metadata

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

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

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

Table 2. Built-in prefixes

Prefix	Description
teiid_rel	Relational Extensions. Uses include function and native query metadata
teiid_sf	Salesforce Extensions.

teiid_mongo	MongoDB Extensions
teiid_odata	OData Extensions
teiid_accumulo	Accumulo Extensions
teiid_excel	Excel Extensions
teiid_ldap	LDAP Extensions
teiid_rest	REST Extensions
teiid_pi	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.

Example: Virtual database DDL

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

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

Example: XML VDB

```
<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 is shown as the domain name.

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

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

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

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[
            -- This procedure produces XML payload
            CREATE VIRTUAL PROCEDURE g1Table(IN p1 integer) RETURNS TABLE (xml_out xml) OPTIONS (UPDATECOUNT 0,
 "teiid_rest:METHOD" 'GET', "teiid_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, "teiid_rest:METHOD" 'GET', "teiid_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 compatibility

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 how SQL is used within Teiid. For details about the exact form of SQL that Teiid accepts, see the BNF for SQL grammar.

Identifiers

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

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

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

Syntax rules

- Identifiers can consist of alphanumeric characters, or the underscore (`_`) character, and must begin with an alphabetic character. Any Unicode character may be used in an identifier.
- Identifiers in double quotes can have any contents. The double quote character can be used if is escaped with an additional double quote; for example, `"some "" id"`
- Because different data sources organize tables in different ways, with some prepending catalog, schema, or user information, 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."sourceschema.sourcetable"` would match the partial name `sourcetable` .

- Columns, column aliases, and schemas cannot contain a dot (`.`) character.
- Identifiers, even when quoted, are not case-sensitive in 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 period (`.`) character, it is a known issue that the alias name will be treated the same as a qualified name and may conflict with fully qualified object names.

Reserved words

Reserved words in Teiid 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. For more information about reserved words, see the *Reserved Keywords* and *Reserved Keywords For Future Use* sections in BNF for SQL grammar.

Expressions

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

You can use following types of expressions in Teiid:

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

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 be any of the 'standard' data types. For information about data types, see Data types.

Syntax rules

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

Some examples of simple literal values are:

```
'abc'
```

Example: Escaped single tick

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

```
5
```

Example: Scientific notation

```
-37.75e01
```

Example: exact numeric type BigDecimal

```
100.0
```

```
true
```

```
false
```

Example: Unicode character

```
'\u0027'
```

Example: Binary

```
X'0F0A'
```

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

Example: Date literal

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

Example: Time literal

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

Example: Timestamp literal

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

Or the ANSI keyword syntax:

Example: Date literal

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

Example: Time literal

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

Example: Timestamp literal

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

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

Aggregate functions

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

You can use the following aggregate functions in Teiid:

COUNT(*)

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

COUNT(x)

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

COUNT_BIG(*)

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

COUNT_BIG(x)

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

SUM(x)

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

AVG(x)

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

MIN(x)

Minimum value in a group (excluding null).

MAX(x)

Maximum value in a group (excluding null).

ANY(x)/SOME(x)

Returns TRUE if any value in the group is TRUE (excluding null).

EVERY(x)

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

VAR_POP(x)

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

VAR_SAMP(x)

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

STDDEV_POP(x)

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

STDDEV_SAMP(x)

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

TEXTAGG(expression [as name], … [DELIMITER char] [QUOTE char | NO QUOTE] [HEADER] [ENCODING id] [ORDER BY …])

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

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

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

Example: Integer value expression

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

could return

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

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

Example: String aggregate expression

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

could return

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

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

Example: List aggregate expression

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

could return

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

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

Syntax rules

- Some aggregate functions may contain a keyword 'DISTINCT' before the expression, indicating that duplicate expression values should be ignored. DISTINCT is not allowed in COUNT(*) and is not meaningful in MIN or MAX (result would be unchanged), so it can be used in COUNT, SUM, and AVG.
- Aggregate functions cannot be used in FROM, GROUP BY, or WHERE clauses without an intervening query expression.
- Aggregate functions cannot be nested within another aggregate function without an intervening query expression.
- Aggregate functions may be nested inside other functions.

- Any aggregate function may take an optional FILTER clause of the form

```
FILTER ( WHERE condition )
```

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

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

Window functions

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

Usage:

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

partition := PARTITION BY expression [, expression]*

frame := range_or_rows extent

range_or_rows := RANGE | ROWS

extent :=
    frameBound
  | BETWEEN frameBound AND frameBound

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

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

Syntax rules

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

Analytical function definitions

Ranking functions

- RANK() – Assigns a number to each unique ordering value within each partition starting at 1, such that the next rank is equal to the count of prior rows.
- DENSE_RANK() – Assigns a number to each unique ordering value within each partition starting at 1, such that the next rank is sequential.
- PERCENT_RANK() – Computed as (RANK - 1) / (RC - 1) where RC is the total row count of the partition.
- CUME_DIST() – Computed as the PR / RC where PR is the rank of the row including peers and RC is the total row count of the partition.

  By default all values are integers - an exception will be thrown if a larger value is needed. Use the system org.teiid.longRanks to have RANK, DENSE_RANK, and ROW_NUMBER return long values instead.

Value functions

- FIRST_VALUE(val) – Return the first value in the window frame with the given ordering.
- LAST_VALUE(val) – Return the last observed value in the window frame with the given ordering.
- LEAD(val [, offset [, default]]) - Access the ordered value in the window that is offset rows ahead of the current row. If there is no such row, then the default value will be returned. If not specified the offset is 1 and the default is null.
- LAG(val [, offset [, default]]) - Access the ordered value in the window that is offset rows behind of the current row. If there is no such row, then the default value will be returned. If not specified the offset is 1 and the default is null.
- NTH_VALUE(val, n) - Returns the nth val in window frame. The index must be greater than 0. If no such value exists, then null is returned.

Row value functions

- ROW_NUMBER() – Sequentially assigns a number to each row in a partition starting at `1`.
- NTILE(n) – Divides the partition into n tiles that differ in size by at most `1`. Larger tiles will be created sequentially starting at the first. `n` must be greater than `0`.

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

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

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

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

Example: Windowed results

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

```
          row_number() over (order by salary) as row_num FROM employees
```

name	salary	max_sal	rank	dense_rank	row_num
John	100000	100000	2	2	2
Henry	50000	50000	5	4	5
John	60000	100000	3	3	3
Suzie	60000	150000	3	3	4
Suzie	150000	150000	1	1	1

Case and searched case

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

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

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

Example case statements

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

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

Scalar subqueries

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

Parameter references

Arrays

Example: Empty arrays

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

Example: Single element array

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

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

Example: General array syntax

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

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

An array element reference takes the form of:

```
array_expr[index_expr]
```

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

Criteria

Criteria can be any of the following items:

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

Usage

```
criteria AND|OR criteria
```

```
NOT criteria
```

```
(criteria)
```

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

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

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

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

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

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

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

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

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

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

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

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

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

Note	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. Additionally, not all sources can use the same regular expression flavor or extensions. In pushdown situations, be careful to ensure that the pattern that you use has the same meaning in 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'`

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

Criteria precedence

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

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

	To prevent lookaheads, the parser does not accept all possible criteria sequences. For example, `a = b is null` is

Note	not accepted, because by the left-associative parsing we first recognize `a =`, then look for a common value expression. `b is null` is not a valid common value expression. Thus, nesting must be used, for example, `(a = b) is null`. For more information about parsing rules, see BNF for SQL grammar.

Scalar functions

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

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

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

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

[a] Non-ASCII range characters or integers used in these functions may produce different results or exceptions depending on where the function is evaluated (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 are not compatible with negative indexing.

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

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

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

TO_BYTES
Return a BLOB from the CLOB with the given encoding.

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

BASE64, HEX, UTF-8-BOM and the builtin Java Charset names are valid values for the encoding [b]. x in a CLOB, encoding is a string, wellformed is a boolean and returns a BLOB. The two argument form defaults to wellformed=true. If wellformed is false, the conversion function will immediately validate the result such that an unmappable character or malformed input will raise an exception. If wellformed is true, then unmappable characters will be replaced by the default replacement character for the character set. Binary formats, such as BASE64 and HEX, will be checked for correctness regardless of the wellformed parameter.

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

REPLACE
Replace all occurrences of a given string with another.

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

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

REGEXP_REPLACE

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

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

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

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

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

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

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

Example regexp_replace

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

Date and time functions

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

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

Function	Definition	Datatype constraint
CURDATE() CURRENT_DATE[()]	Return current date - will return the same value for all invocations in the user command.	returns date.
CURTIME()	Return current time - will return the same value for all invocations in the user command. See also CURRENT_TIME.	returns time
NOW()	Return current timestamp (date and time with millisecond precision) - will return the same value for all invocations in the user command or procedure instruction. See also CURRENT_TIMESTAMP.	returns timestamp
CURRENT_TIME[(precision)]	Return current time - will return the same value for all invocations in the user command. The 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},

EPOCH(x)	Return seconds since the unix epoch with microsecond precision	x in {date, timestamp}, returns double
EXTRACT(YEAR|MONTH|DAY |HOUR|MINUTE|SECOND|QUARTER|EPOCH 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, QUARTER, EPOCH functions 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}, epoch returns double, the others return 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. 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. For customization information, see *System Properties* in the Administrator's Guide.	x in {date, timestamp}, returns integer
YEAR(x)	Return four-digit year	x in {date, timestamp}, returns integer

Timestampadd

Add a specified interval amount to the timestamp.

Syntax

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

Arguments

Name	Description
interval	A datetime interval unit, can be one of the following keywords: • SQL_TSI_FRAC_SECOND - fractional seconds (billionths of a second) • SQL_TSI_SECOND - seconds • SQL_TSI_MINUTE - minutes • SQL_TSI_HOUR - hours • SQL_TSI_DAY - days

	• SQL_TSI_WEEK - weeks using Sunday as the first day • SQL_TSI_MONTH - months • SQL_TSI_QUARTER - quarters (3 months) where the first quarter is months 1-3, etc. • SQL_TSI_YEAR - years
count	A long or integer count of units to add to the timestamp. Negative values will subtract that number of units. Long values are allowed for symmetry with TIMESTAMPDIFF - but the effective range is still limited to integer values.
timestamp	A datetime expression.

Example

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

Timestampdiff

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

Syntax

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

Arguments

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

Example

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

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

Compatibility issues

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

Parsing date datatypes from strings

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. For more information about how this class defines date and time string formats, see Javadocs for SimpleDateFormat. Note that the format strings are specific to your Java default locale.

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

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

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

For example: GMT-05:00

Type conversion functions

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

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

Within each function call, you include the following arguments:

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

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

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

When the 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. For more information about caching in the Lookup function, see the Caching Guide .

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

System functions

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

COMMANDPAYLOAD
Retrieve a string from the command payload or null if no command payload was specified. The command payload is set by the `TeiidStatement.setPayload` method on the 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 included 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 last insert statement of this session returning a generated key.

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

`GENERATED_KEY()`

The return value is long.

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

``GENERATED_KEY(column_name)` ``

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

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

XML functions

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

Sample data for examples
Examples provided with XML functions use the following table structure

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

with Data

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

XMLCAST
Cast to or from XML.

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

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

XMLCOMMENT
Returns an XML comment.

```
XMLCOMMENT(comment)
```

Comment is a string. Return value is XML.

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

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

Content is XML. Return value is XML.

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

Concatenate two or more XML fragments

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

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

XMLELEMENT

Returns an XML element with the given name and content.

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

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

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

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

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

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

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

Simple example

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

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

Multiple columns

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

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

Columns as attributes

```
SELECT XMLELEMENT("customer",
          XMLELEMENT("name", c.CustomerName,
            XMLATTRIBUTES(
                  "contact" as c.ContactName,
```

```
                    "id" as c.CustomerID
              )
            )
         )
FROM   Customer c
WHERE  c.CustomerID = 87;

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

XMLFOREST

Returns an concatenation of XML elements for each content item.

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

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

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

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

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

Example

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

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

XMLAGG

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

```
XMLAGG(xml)
```

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

Example

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


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

```
<customer><name>Wellington Importadora</name><contact>Paula Parente</contact></customer>
<customer><name>White Clover Markets</name><contact>Karl Jablonski</contact></customer>
</customers>
```

XMLPARSE

Returns an XML type representation of the string value expression.

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

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

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

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

```
SELECT XMLPARSE(CONTENT '<customer><name>Wartian Herkku</name><contact>Pirkko Koskitalo</contact></customer>' 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]]*
```

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

Namespaces may also be directly declared in the xquery prolog.

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

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

xquery in string. Return value is XML.

XMLQUERY is part of the SQL/XML 2006 specification.

For more information, see XMLTABLE in FROM clause.

Note	See also XQuery optimization.

XMLEXISTS

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

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

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

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

Namespaces can also be directly declared in the xquery prolog.

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

xquery in string. Return value is boolean.

XMLEXISTS is part of the SQL/XML 2006 specification.

Note	See also XQuery optimization.

XMLSERIALIZE
Returns a character type representation of the XML expression.

```
XMLSERIALIZE([(DOCUMENT|CONTENT)] xml [AS datatype] [ENCODING enc] [VERSION ver] [(INCLUDING|EXCLUDING) 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. For more information, see *XMLQUERY* in XML functions.

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

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

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

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

Sample XML for xpathValue Ignoring Namespaces

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

Function:

Sample xpathValue Ignoring Namespaces

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

Results in `Hello World`

Example: Generating hierarchical XML from flat data structure
With following table and its contents

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

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

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

use the SQL statement in 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
```

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

JSON functions

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

Sample data for examples
Examples provided with XML functions use the following table structure:

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

with Data

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

JSONARRAY
Returns a JSON array.

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

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

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

mixed value example

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

Would return

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

Using JSONARRAY on a Table

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

```
[89,"White Clover Markets"]
```

JSONOBJECT
Returns a JSON object.

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

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

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

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

mixed value example

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

Would return

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

Using JSONOBJECT on a Table

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

Another example

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

Another example

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

JSONPARSE
Validates and returns a JSON result.

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

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

A null for either input will return null.

JSON parse of a simple literal value

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

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

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

JSONTOXML
Returns an XML document from JSON.

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

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

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

The result is always a well-formed XML document.

The mapping to XML uses the following rules:

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

JSON:

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

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

XML:

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

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

JSON:

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

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

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

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

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

JSON:

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

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

XML:

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

```
<?xml version="1.0" ?>
<root>
```

```
  <_x002F_invalid>abc</_x002F_invalid>
</root>
```

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

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

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

For more information, see JSONTABLE.

JSONPATHVALUE
Extracts a single JSON value as a string.

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

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

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

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

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

Would return

```
null
```

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

Would return

```
value1
```

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

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

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

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

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

Would return

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

Security functions

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

HASROLE
Whether the current caller has the 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. Foreign/JAAS roles/groups names are not valid for this function, unless there is corresponding data role with the same name.

MD5
Computes the MD5 hash of the value.

```
MD5(value)
```

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

SHA1
Computes the SHA-1 hash of the value.

```
SHA1(value)
```

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

SHA2_256
Computes the SHA-2 256 bit hash of the value.

```
SHA2_256(value)
```

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

SHA2_512
Computes the SHA-2 512 bit hash of the value.

```
SHA2_512(value)
```

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

AES_ENCRYPT

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

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

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

AES_DECRYPT

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

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

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

Spatial functions

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

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

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

ST_GeomFromText
Returns a geometry from a Clob in WKT format.

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

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

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

```
ST_GeogFromText(text)
```

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

ST_GeomFromWKB/ST_GeomFromBinary
Returns a geometry from a BLOB in WKB format.

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

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

ST_GeomFromEWKB
Returns a geometry from a BLOB in EWKB format.

```
ST_GeomFromEWKB(bin)
```

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

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

```
ST_GeomFromEWKB(bin)
```

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

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

```
ST_GeomFromEWKT(text)
```

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

ST_GeomFromGeoJSON

Returns a geometry from a CLOB in GeoJSON format.

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

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

ST_GeomFromGML

Returns a geometry from a CLOB in GML2 format.

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

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

ST_AsText

```
ST_AsText(geom)
```

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

ST_AsBinary

```
ST_AsBinary(geo)
```

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

ST_AsEWKB

```
ST_AsEWKB(geom)
```

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

ST_AsGeoJSON

```
ST_AsGeoJSON(geom)
```

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

ST_AsGML

```
ST_AsGML(geom)
```

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

ST_AsEWKT

```
ST_AsEWKT(geo)
```

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

ST_AsKML

```
ST_AsKML(geom)
```

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

&&

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

```
geom1 && geom2
```

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

ST_Contains

Returns true if `geom1` contains `geom2` .

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

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

ST_Crosses

Returns true if the geometries cross.

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

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

ST_Disjoint

Returns true if the geometries are disjoint.

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

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

ST_Distance

Returns the distance between two geometries.

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

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

ST_DWithin

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

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

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

ST_Equals

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

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

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

ST_Intersects

Returns true if the geometries intersect.

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

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

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

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

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

ST_Overlaps
Returns true if the geometries overlap.

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

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

ST_Relate
Test or return the intersection of geom1 and geom2.

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

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

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

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

ST_Touches
Returns true if the geometries touch.

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

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

ST_Within
Returns true if `geom1` is completely inside `geom2` .

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

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

ST_Area
Returns the area of geom.

```
ST_Area(geom)
```

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

ST_CoordDim
Returns the coordinate dimensions of geom.

```
ST_CoordDim(geom)
```

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

ST_Dimension

Returns the dimension of geom.

```
ST_Dimension(geom)
```

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

ST_EndPoint

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

```
ST_EndPoint(geom)
```

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

ST_ExteriorRing

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

```
ST_ExteriorRing(geom)
```

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

ST_GeometryN

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

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

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

ST_GeometryType

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

```
ST_GeometryType(geom)
```

`geom` is a geometry. Return value is a string.

ST_HasArc

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

```
ST_HasArc(geom)
```

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

ST_InteriorRingN

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

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

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

ST_IsClosed

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

```
ST_IsClosed(geom)
```

`geom` is a geometry. Return value is a Boolean.

ST_IsEmpty

Returns true if the set of points is empty.

```
ST_IsEmpty(geom)
```

`geom` is a geometry. Return value is a Boolean.

ST_IsRing

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

```
ST_IsRing(geom)
```

`geom` is a geometry. Return value is a Boolean.

ST_IsSimple

Returns true if the `geom` is simple.

```
ST_IsSimple(geom)
```

`geom` is a geometry. Return value is a Boolean.

ST_IsValid

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

```
ST_IsValid(geom)
```

`geom` is a geometry. Return value is a Boolean.

ST_Length

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

```
ST_Length(geo)
```

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

ST_NumGeometries

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

```
ST_NumGeometries(geom)
```

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

ST_NumInteriorRings

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

```
ST_NumInteriorRings(geom)
```

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

ST_NunPoints

Returns the number of points in `geom`.

```
ST_NunPoints(geom)
```

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

ST_PointOnSurface

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

```
ST_PointOnSurface(geom)
```

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

ST_Perimeter

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

```
ST_Perimeter(geom)
```

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

ST_PointN

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

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

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

ST_SRID

Returns the SRID for the geometry.

```
ST_SRID(geo)
```

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

ST_SetSRID

Set the SRID for the given geometry.

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

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

ST_StartPoint

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

```
ST_StartPoint(geom)
```

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

ST_X

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

```
ST_X(geom)
```

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

ST_Y

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

```
ST_Y(geom)
```

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

ST_Z

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

```
ST_Z(geom)
```

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

ST_Boundary

Computes the boundary of the given geometry.

```
ST_Boundary(geom)
```

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

ST_Buffer

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

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

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

ST_Centroid

Computes the geometric center point of geom.

```
ST_Centroid(geom)
```

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

ST_ConvexHull

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

```
ST_ConvexHull(geom)
```

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

ST_CurveToLine

Converts a CircularString/CurvedPolygon to a LineString/Polygon. Not currently implemented in 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 geometries. Return value is a geometry.

ST_Envelope

Computes the 2D bounding box of the given geometry.

```
ST_Envelope(geom)
```

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

ST_Force_2D

Removes the z coordinate value if present.

```
ST_Force_2D(geom)
```

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

ST_Intersection

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

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

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

ST_Simplify

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

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

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

ST_SimplifyPreserveTopology

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

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

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

ST_SnapToGrid

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

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

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

ST_SymDifference

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

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

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

ST_Transform
Transforms the geometry value from one coordinate system to another.

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

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

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

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

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

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

```
ST_Extent(geom)
```

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

ST_Point
Retuns the Point for the given coordinates.

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

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

ST_Polygon
Returns the Polygon with the given shell and SRID.

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

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

Miscellaneous functions

Documents additional functions and those contributed by other projects.

array_get
Returns the object value at a given array index.

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

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

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

array_length
Returns the length for a given array.

```
array_length(array)
```

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

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

uuid
Returns a universally unique identifier.

```
uuid()
```

The return type is string.

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

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

osdq.random
Returns the randomized string. For example, `jboss teiid` may randomize to `jtids soibe`.

```
random(sourceValue)
```

The `sourceValue` is the string to be randomized.

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

```
digit(sourceValue)
```

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

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

```
whitespaceIndex(sourceValue)
```

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

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

```
validCreditCard(cc)
```

`cc` is the credit card number string to check.

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

```
validSSN(ssn)
```

`ssn` is the social security number string to check.

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

```
validPhone(phone)
```

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

osdq.validEmail
Check whether an email address is valid. Returns `true` if valid.

```
validEmail(email)
```

`email` is the email address string to check.

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

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

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

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

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

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

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

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

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

osdq.levenshteinDistance

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

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

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

osdq.intersectionFuzzy

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

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

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

osdq.minusFuzzy

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

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

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

osdq.unionFuzzy

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

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

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

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.

Deterministic

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

User Deterministic

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

Session Deterministic

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

Command Deterministic

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

Nondeterministic

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

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

DML commands

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

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

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

Set operations

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

Usage:

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

Syntax Rules:

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

SELECT command

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

A SELECT command can contain the following clauses:

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

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

Stage 1: WITH clause

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

Stage 2: FROM clause

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

Stage 3: WHERE clause

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

Stage 4: GROUP BY clause

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

Stage 5: HAVING clause

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

Stage 6: SELECT clause

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

Stage 7: ORDER BY clause

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

Stage 8: LIMIT clause

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

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

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

VALUES command

The VALUES command is used to construct a simple table.

Example syntax

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

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

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

Update commands

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

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 results in 1 or more records being updated, or in no records being updated if none match the criteria.

Example syntax

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

DELETE command

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

Example syntax

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

UPSERT (MERGE) command

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

Example syntax

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

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

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

EXECUTE command

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

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

- EXEC
- CALL

Example syntax

```
EXECUTE proc()
```

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

Named parameter syntax

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

Syntax rules

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

Procedural relational command

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

Example syntax

```
select * from proc
```

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

```
select output_param1, output_param2 from proc, table where input_param1 = table.col1 and input_param2 = table.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 are projected as two columns:
    - One to represents the output value.
    - One with the name {column name}_IN that represents the input of the parameter.
- Input values are passed via criteria. Values can be passed by `=`, `is null`, or as `in` predicates. Disjuncts are not allowed. It is also not possible to pass the value of a non-comparable column through an equality predicate.
- The procedure view automatically has an access pattern on its IN and IN_OUT parameters. The access pattern allows the procedure view to be planned correctly as a dependent join when necessary, or to fail when sufficient criteria cannot be found.
- Procedures that contain duplicate names between the parameters (IN, IN_OUT, OUT, RETURN) and the result set columns cannot be used in a procedural relational command.
- If there is already a table or view with the same name as the procedure, then it cannot be invoked via procedural relational syntax.
- Default values for IN or IN_OUT parameters are not used if there is no criteria present for a given input. Default values are only valid for named procedure syntax. For more information, see EXECUTE.

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

Multiple execution

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

Anonymous procedure block

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

Example syntax

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

Syntax rules

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

Subqueries

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

Subquery types:

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

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

Example subquery in FROM clause (inline view)

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

Subqueries can appear anywhere where an expression or criteria is expected.
You can use subqueries in quantified criteria, the `EXISTS` predicate, the `IN` predicate, and as Scalar subqueries.

Example subquery in WHERE using EXISTS

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

Example quantified comparison subqueries

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

Example IN subquery

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

See also Subquery Optimization.

WITH clause

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

Usage

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

Syntax rules

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

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

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

Examples

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

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

Recursive common table expressions

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

Usage

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

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

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

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

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

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

SELECT clause

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

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 tables for SELECT, UPDATE, and DELETE statements.

Example Syntax:

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

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

Example hint

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

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

MAKEIND

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

MAKEDEP

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

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

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

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

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

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

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

The PRESERVE hint can be used against an ANSI join tree to preserve the structure of the join, rather than allowing the 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 tables

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

A nested table can have correlated references to preceding `FROM` clause column references as long as `INNER` and `LEFT OUTER` joins are used. This is especially useful in cases where then nested expression is a procedure or function call.

Valid nested table example

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

Invalid nested table example
The following nested table example is invalid, because `t1` appears after the nested table in the FROM clause:

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

Note	Multiple execution Using a correlated nested table can result in multiple executions of the table expression — one for each correlated row.

XMLTABLE

The XMLTABLE function uses XQuery to produce tabular output. The XMLTABLE function is implicitly a nested table and it can be used within FROM clauses. XMLTABLE is part of the SQL/XML 2006 specification.

Usage

```
XMLTABLE([<NSP>,] xquery-expression [<PASSING>] [COLUMNS <COLUMN>, ... ]) AS name
```

```
COLUMN := name (FOR ORDINALITY | (datatype [DEFAULT expression] [PATH string]))
```

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

Note	See also XQuery optimization.

Parameters

- The optional XMLNAMESPACES clause specifies the namepaces that you can use in the XQuery and COLUMN path expressions.
- The xquery-expression must be a valid XQuery. Each sequence item returned by the xquery is used to create a row of values as defined by the COLUMNS clause.
- If COLUMNS is not specified, that is equivalent to a COLUMNS clause that returns the entire item as an XML value, as in the following example:

  ```
  "COLUMNS OBJECT_VALUE XML PATH '.'"
  ```

- FOR ORDINALITY columns are typed as integers and return 1-based item numbers as their value.
- Each non-ordinality column specifies a type, and optionally specifies a PATH and a DEFAULT expression.
- If PATH is not specified, then the path is the same as the column name.

Syntax Rules

- You can specify only 1 FOR ORDINALITY column.
- Columns names must not contain duplicates.
- You can use binary large object (BLOB) datatypes, but there is built-in compatibility only for `xs:hexBinary` values. For xs:base64Binary, use a workaround of a PATH that uses the following explicit value constructor: `xs:base64Binary(<path>)` .
- The column expression must evaluate to a single value if a non-array type is expected.
- If an array type is specified, then an array is returned, unless there are no elements in the sequence, in which case a null value is returned.
- An empty element is not a valid null value, because its value is effectively an empty string. Use the `xsi:nil` attribute to specify a null value for an element.

XMLTABLE examples

Use of PASSING, returns 1 row [1]

```
select * from xmltable('/a' PASSING xmlparse(document '<a id="1"/>') COLUMNS id integer PATH '@id') x
```

As a nested table

```
select x.* from t, xmltable('/x/y' PASSING t.doc COLUMNS first string, second FOR ORDINALITY) x
```

Invalid multi-value

```
select * from xmltable('/a' PASSING xmlparse(document '<a><b id="1"/><b id="2"/></a>') COLUMNS id integer PATH
'b/@id') x
```

Array multi-value

```
select * from xmltable('/a' PASSING xmlparse(document '<a><b id="1"/><b id="2"/></a>') COLUMNS id integer[] PATH
 'b/@id') x
```

Nil element

```
select * from xmltable('/a' PASSING xmlparse(document '<a xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
><b xsi:nil="true"/></a>') COLUMNS id integer PATH 'b') x
```

Note	In the preceding example, an exception would be thrown if the `nil` attribute (`xsi:nil="true"`) were not specified, converting `b` to an integer value.

ARRAYTABLE

The ARRAYTABLE function processes an array input to produce tabular output. The function itself defines what columns it projects. The ARRAYTABLE function is implicitly a nested table and can be used within FROM clauses.

Usage

```
ARRAYTABLE([ROW|ROWS] expression COLUMNS <COLUMN>, ...) AS name
COLUMN := name datatype
```

Parameters

expression
The array to process, which should be a java.sql.Array or java array value.

ROW|ROWS
If ROW (the default) is specified, then only a single row is produced from the given array (typically a one dimensional array). If ROWS is specified, then multiple rows are produced. A multidimensional array is expected, and one row is produced for every non-null element of the outer array.

If the expression is null, no rows are produced.

Syntax rules

- Columns names cannot contain duplicates.

Array table examples

- As a nested table:

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

ARRAYTABLE is effectively a shortcut for using the `array_get` function in a nested table.

For example, the following ARRAYTABLE function:

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

is the same as the following statement which uses an `array_get` function:

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

OBJECTTABLE

The OBJECTTABLE function processes an object input to produce tabular output. The function itself defines what columns it projects. The OBJECTTABLE function is implicitly a nested table and can be used within FROM clauses.

Usage

```
OBJECTTABLE([LANGUAGE lang] rowScript [PASSING val AS name ...] COLUMNS colName colType colScript [DEFAULT 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 'teiid_script' is used. name:: An identifier that binds the `val` expression value into the script context. rowScript:: A string literal that specifies the script to create the row values. For each non-null item that the Iterator produces, the columns are evaluated. colName/colType:: ID/data type of the column, which can optionally be defaulted with the DEFAULT clause expression `defaultExpr`. colScript:: A string literal that specifies the script that evaluates to the column value.

Syntax rules

- Columns names cannot contain duplicates.
- Teiid places several special variables in the script execution context. The CommandContext is available as `teiid_context`. Additionally the `colScripts` may access `teiid_row` and `teiid_row_number`. `teiid_row` is the current row object produced by the row script. `teiid_row_number` is the current 1-based row number.
- `rowScript` is evaluated to an Iterator. If the results is already an Iterator, it is used directly. If the evaluation result is an Iteratable, Array, or Array type, then an Iterator is obtained. Any other Object will be treated as an Iterator of a single item. In all cases null row values are skipped.

Note	Although there are no restrictions on naming PASSING variables, it is best to choose names that you can reference as identifiers in the target language.

OBJECTTABLE examples

- Accessing special variables:

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

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

Note	Languages other than teiid_script generally permit unrestricted access to Java functionality. As a result, by default, their use is restricted. You can override the restrictions by declaring allowable languages by name in the `allowed-languages` property. To use OBJECTTABLE, even from within view definitions that are not normally subject to permission checks, you must define the `allowed-languages` property. You must also set language access rights for user accounts to enable users to process OBJECTTABLE functions. • For more information about about teiid_script, see the next section. • For more information about enabling the use of languages other than teiid_script, see *allowed-languages* in Virtual database properties.

- For more information about setting user account permission, see *User query permissions* in Permissions.

teiid_script

teiid_script is a simple scripting expression language that allows access to passing and special variables, and to non-void 0-argument methods on objects and indexed values on arrays/lists. A teiid_script expression begins by referencing the passing or special variable. Then, any number of `.` accessors can be chained to evaluate the expression to a different value. Methods may be accessed by their property names, for example, foo rather than getFoo. If the object includes both a `getFoo()` and `foo()` method, then the accessor `foo` references `foo ()`, and `getFoo` should be used to call the getter. An array or list index is accessed using a 1-based positive integral value, using the same `.` accessor syntax. The same logic as the system function `array_get` is used. That is, if the index is out of bounds, `null` is returned, rather than an exception.

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

teiid_script examples

- To get the VDB description string:

```
teiid_context.session.vdb.description
```

- To get the first character of the VDB description string:

```
teiid_context.session.vdb.description.toCharArray.1
```

TEXTTABLE

The TEXTTABLE function processes character input to produce tabular output. It provides both fixed and delimited file format parsing. The function itself defines what columns it projects. The TEXTTABLE function is implicitly a nested table and can be used within FROM clauses.

Usage

```
TEXTTABLE(expression [SELECTOR string] COLUMNS <COLUMN>, ... [NO ROW DELIMITER | ROW DELIMITER char] [DELIMITER
 char] [(QUOTE|ESCAPE) char] [HEADER [integer]] [SKIP integer] [NO TRIM]) AS name
```

Where <COLUMN>

```
COLUMN := name (FOR ORDINALITY | ([HEADER string] datatype [WIDTH integer [NO TRIM]] [SELECTOR string integer]))
```

Parameters

expression
The text content to process, which should be convertible to a character large object (CLOB).

SELECTOR
Used with files containing multiple types of rows (example: order header, detail, summary). A TEXTTABLE SELECTOR specifies which lines to include in the output. Matching lines must begin with the selector string. The selector in column delimited files must be followed by the column delimiter.

If a TEXTTABLE SELECTOR is specified, a SELECTOR may also be specified for column values. A column SELECTOR argument will select the nearest preceding text line with the given SELECTOR prefix, and select the value at the given 1-based integer position (which includes the selector itself). If no such text line or position with a given line exists, a null value will be produced. A column SELECTOR is not valid with fixed width parsing.

NO ROW DELIMITER
Specifies that fixed parsing should not assume the presence of newline row delimiters.

ROW DELIMITER
Sets the row delimiter / newline to an alternate character. Defaults to the new-line character - with built-in handling for treating carriage return newline as a single character. If ROW DELIMITER is specified, carriage return is given no special treatment.

DELIMITER
Sets the field delimiter character to use. Defaults to `,`.

QUOTE
Sets the quote, or qualifier, character used to wrap field values. Defaults to `"`.

ESCAPE
Sets the escape character to use if no quoting character is in use. This is used in situations where the delimiter or new line characters are escaped with a preceding character, e.g. `\`.

HEADER
Specifies the text line number (counting every new line) on which the column names occur. If the HEADER option for a column is specified, then that will be used as the expected header name. All lines prior to the header will be skipped. If HEADER is specified, then the header line will be used to determine the TEXTTABLE column position by case-insensitive

name matching. This is especially useful in situations where only a subset of the columns are needed. If the HEADER value is not specified, it defaults to 1. If HEADER is not specified, then columns are expected to match positionally with the text contents.

SKIP

Specifies the number of text lines (counting every new line) to skip before parsing the contents. HEADER can be specified with SKIP.

FOR ORDINALITY

Column that is typed as integer and returns a 1-based item number as its value.

WIDTH

Indicates the fixed-width length of a column in characters, not bytes. With the default ROW DELIMITER, a CR NL sequence counts as a single character.

NO TRIM

When specified on a TEXTTABLE, it affects all column and header values. When NO TRIM is specified on a column, the fixed or unqualified text value is not trimmed of leading and trailing white space.

Syntax Rules

- If width is specified for one column it must be specified for all columns and be a non-negative integer.
- If width is specified, then fixed width parsing is used, and ESCAPE, QUOTE, column SELECTOR, nor HEADER should not be specified.
- If width is not specified, then NO ROW DELIMITER cannot be used.
- Columns names must not contain duplicates.
- The characters specified for QUOTE, DELIMITER, and ROW DELIMITER must all be different.

TEXTTABLE examples

- Use of the HEADER parameter, returns 1 row ['b']:

```
SELECT * FROM TEXTTABLE(UNESCAPE('col1,col2,col3\na,b,c') COLUMNS col2 string HEADER) x
```

- Use of fixed width, returns 2 rows ['a', 'b', 'c'], ['d', 'e', 'f']:

```
SELECT * FROM TEXTTABLE(UNESCAPE('abc\ndef') COLUMNS col1 string width 1, col2 string width 1, col3 string 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
```

JSONTABLE

The JSONTABLE function uses JsonPath to produce tabular output. The JSONTABLE function is implicitly a nested table and can be used within FROM clauses.

Usage

```
JSONTABLE(value, path [, nullLeafOnMissing] COLUMNS <COLUMN>, ... ) AS name
```

```
COLUMN := name (FOR ORDINALITY | (datatype [PATH string]))
```

See also JsonPath

Parameters

value
A clob containing a valid JSON document.

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

PATH
String should be a valid JsonPath. If an array value is returned, then each non-null element will be used to generate a row. Otherwise a single non-null item will be used to create a single row.

FOR ORDINALITY
Column typed as integer. Returns a 1-based item number as its value.

- Each non-ordinality column specifies a type and optionally a PATH.
- If PATH is not specified, then the path will be generated from the column name: @['name'], which will look for an object key value matching name. If PATH is specified, it must begin with @, which means that the path will be processed relative the the current row context item.

Syntax Rules

- Columns names must not contain duplicates.
- You cannot use array types with the JSONTABLE function.

JSONTABLE examples
Use of passing, returns 1 row [1]:

```
select * from jsontable('{"a": {"id":1}}}', '$.a' COLUMNS id integer) x
```

As a nested table:

```
select x.* from t, jsontable(t.doc, '$.x.y' COLUMNS first string, second FOR ORDINALITY) x
```

With more complicated paths:

```
select x.* from jsontable('[{"firstName": "John", "lastName": "Wayne", "children": []}, {"firstName": "John", "
lastName": "Adams", "children":["Sue","Bob"]}]', '$.*' COLUMNS familyName string path '@.lastName', children in
teger path '@.children.length()' ) x
```

Differences with XMLTABLE

Processing of JSON to tabular results was previously recommended through the use of XMLTABLE with JSONTOXML. For most tasks, JSONTABLE provides a simpler syntax. However, there are some differences to consider:

- JSONTABLE parses the JSON completely, the processes it. XMLTABLE uses streaming processing to reduce the memory overhead.
- JsonPath is not as powerful as XQuery. There are a lot of functions and operations available in XQuery/XPath that are not available in JsonPath.
- JsonPath does not allow for parent references in the column paths. There is no ability to reference the root or any part of the parent hierarchy (.. in XPath).

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 is returned for each group, after optionally filtering those aggregate rows based on a HAVING clause.

The general form of the GROUP BY is:

```
GROUP BY expression [,expression]*
```

```
GROUP BY ROLLUP(expression [,expression]*)
```

Syntax Rules

- Column references in the group by cannot be made to alias names in the SELECT clause.
- Expressions used in the group by must appear in the select clause.
- Column references and expressions in the SELECT/HAVING/ORDER BY clauses that are not used in the group by clause must appear in aggregate functions.
- If an aggregate function is used in the SELECT clause and no GROUP BY is specified, an implicit GROUP BY will be performed with the entire result set as a single group. In this case, every column in the SELECT must be an aggregate function as no other column value will be fixed across the entire group.
- The GROUP BY columns must be of a comparable type.

Rollups

Just like normal grouping, ROLLUP processing logically occurs before the HAVING clause is processed. A ROLLUP of expressions will produce the same output as a regular grouping with the addition of aggregate values computed at higher aggregation levels. For N expressions in the ROLLUP, aggregates will be provided over (), (expr1), (expr1, expr2), etc. up to (expr1, … exprN-1), with the other grouping expressions in the output as null values. The following example uses a normal aggregation query:

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

The query returns the following data:

Table 1. Data returned by a normal aggregation query

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

In contrast, the following example uses a rollup query:

Data returned from a rollup query

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

would return:

country	city	sum(amount)
US	St. Louis	10000
US	Raleigh	150000
US	Denver	20000
US	**<null>**	180000
UK	Birmingham	50000
UK	London	75000
UK	**<null>**	125000
<null>	**<null>**	305000

Note	Not all sources are compatible with ROLLUPs, and compared to normal aggregate processing, some optimizations might be inhibited by the use of a ROLLUP.

The use of ROLLUPs in Teiid is currently limited in comparison to the SQL specification.

HAVING Clause

The `HAVING` clause operates exactly as a `WHERE` clause, although it operates on the output of a `GROUP BY`. You can use the same syntax with the `HAVING` clause as with the `WHERE` clause.

Syntax Rules

- Expressions used in the GROUP BY clause must contain either an aggregate function (`COUNT` , `AVG` , `SUM` , `MIN` , `MAX`), or be one of the grouping expressions.

ORDER BY clause

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

Usage

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

Syntax rules

- Sort columns can be specified positionally by a 1-based positional integer, by SELECT clause alias name, by SELECT clause expression, or by an unrelated expression.
- Column references can appear in the SELECT clause as the expression for an aliased column, or can reference columns from tables in the FROM clause. If the column reference is not in the SELECT clause, the query cannot be a set operation, specify SELECT DISTINCT, or contain a GROUP BY clause.
- Unrelated expressions, expressions not appearing as an aliased expression in the select clause, are allowed in the ORDER BY clause of a non-set QUERY. The columns referenced in the expression must come from the from clause table references. The column references cannot be to alias names or positional.
- The ORDER BY columns must be of a comparable type.
- If an ORDER BY is used in an inline view or view definition without a LIMIT clause, it is removed by the 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 the default internal sorting behavior for Teiid. However, not all sources return results with nulls sorted as low values by default, and Teiid might return results with different null orderings.

Warning	The use of positional ordering is no longer supported by the ANSI SQL standard and is a deprecated feature in Teiid. It is best 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. YOu can specify an optional offset (the number of rows to skip). The LIMIT clause can also be specified using the SQL 2008 OFFSET/FETCH FIRST clauses. If an ORDER BY is also specified, it will be applied before the OFFSET/LIMIT are applied. If an ORDER BY is not specified there is generally no guarantee what subset of rows will be returned.

Usage

```
LIMIT [offset,] limit
```

```
LIMIT limit OFFSET offset
```

```
[OFFSET offset ROW|ROWS] [FETCH FIRST|NEXT [limit] ROW|ROWS ONLY]
```

Syntax rules

- The LIMIT/OFFSET expressions must be a non-negative integer or a parameter reference (`?`). An offset of `0` is ignored. A limit of `0` returns no rows.
- The terms FIRST/NEXT are interchangeable as well as ROW/ROWS.
- The LIMIT clause can take an optional preceding NON_STRICT hint to indicate that push operations should not be inhibited, even if the results will not be consistent with the logical application of the limit. The hint is only needed on unordered limits, for example, `"SELECT * FROM VW /*+ NON_STRICT */ LIMIT 2"`.

LIMIT clause examples

- `LIMIT 100` returns the first 100 records (rows 1-100).
- `LIMIT 500, 100` skips 500 records and returns the next 100 records(rows 501-600).
- `OFFSET 500 ROWS` skips 500 records.
- `OFFSET 500 ROWS FETCH NEXT 100 ROWS ONLY` skips 500 records and returns the next 100 records (rows 501-600).
- `FETCH FIRST ROW ONLY` returns only the first record.

INTO clause

Warning	Usage of the INTO Clause for inserting into a table has been been deprecated. An INSERT with a query command should be used instead. For information about using INSERT, see INSERT command.

When the into clause is specified with a SELECT, the results of the query are inserted into the specified table. This is often used to insert records into a temporary table. The INTO clause immediately precedes the FROM clause.

Usage

```
INTO table FROM ...
```

Syntax rules

- The `INTO` clause is logically applied last in processing, after the `ORDER BY` and `LIMIT` clauses.
- Teiid's support for `SELECT INTO` is similar to Microsoft SQL Server. The target of the `INTO` clause is a table where the result of the `SELECT` command will be inserted.

  For example, the following statement:

  ```
  SELECT col1, col2 INTO targetTable FROM sourceTable
  ```

  inserts `col1` and `col2` from the `sourceTable` into the `targetTable`.
- You cannot combine SELECT INTO with a UNION query.

  That is, you cannot select the results from a `sourceTable UNION` query for insertion into a `targetTable`.

OPTION clause

The OPTION keyword denotes options that a user can pass in with a command. These options are specific to Teiid and are not covered by any SQL specification.

Usage

```
OPTION option (, option)*
```

Supported options

`MAKEDEP table (,table)*`
Specifies source tables that should be made dependent in the join.

`MAKEIND table (,table)*`
Specifies source tables that should be made independent in the join.

`MAKENOTDEP table (,table)*`
Prevents a dependent join from being used.

`NOCACHE [table (,table)*]`
Prevents cache from being used for all tables or for the given tables.

Examples

```
OPTION MAKEDEP table1
```

```
OPTION NOCACHE
```

All tables specified in the OPTION clause should be fully qualified. However, the table name can match either the fully qualified name or an alias name.

The MAKEDEP and MAKEIND hints can take optional arguments to control the dependent join. The extended hint form is:

```
MAKEDEP tbl([max:val] [[no] join])
```

- `tbl(JOIN)` means that the entire join should be pushed.
- `tbl(NO JOIN)` means that the entire join should not be pushed.
- `tbl(MAX:val)` means that the dependent join should only be performed if there are less than the maximum number of values from the independent side.

Tip	Teiid does not accept PLANONLY, DEBUG, and SHOWPLAN arguments in the OPTION clause. For information about how to perform the functions formerly provided by these options, see the Client Developer’s Guide.

Note	MAKEDEP and MAKENOTDEP hints can take table names in the form of `@view1.view2…table` . For example, with an inline view `"SELECT * FROM (SELECT * FROM tbl1, tbl2 WHERE tbl1.c1 = tbl2.c2) AS v1 OPTION MAKEDEP @v1.tbl1"` the hint will now be understood as applying under the v1 view.

DDL Commands

Teiid is compatible with a subset of the DDL commands for creating or dropping temporary tables and manipulating procedure and view definitions at runtime. It is not currently possible to arbitrarily drop or create non-temporary metadata entries. For information about the DDL statements that you can use to define schemas in a virtual database, see DDL metadata.

To make non-temporary metadata updates persistent, you must configure a `MetadataRepository`. For more information, see *Runtime Metadata Updates* in the Developer's Guide.

Temporary Tables

You can create and use temporary (*temp*) tables in Teiid. Temporary tables are created dynamically, but are treated as any other physical table.

Local temporary tables

Local temporary tables can be defined implicitly by referencing them in a INSERT statement or explicitly with a CREATE TABLE statement. Implicitly created temp tables must have a name that starts with `#`.

Note	Teiid interprets *local* to mean that a temporary table is scoped to the session or block of the virtual procedure that creates it. This interpretation differs from the SQL specification and from the interpretation that other database vendors implement. After exiting a block or at the termination of a session, the table is dropped. Session tables and other temporary tables that a calling procedures creates are not visible to called procedures. If a temporary table of the same name is created in a called procedure, then a new instance is created.

Creation syntax

You can create local temporary tables explicitly or implicitly.

Explicit creation syntax

Local temporary tables can be defined explicitly with a CREATE TABLE statement, as in the following example:name: value

```
CREATE LOCAL TEMPORARY TABLE name (column type [NOT NULL], ... [PRIMARY KEY (column, ...)]) [ON COMMIT PRESE
RVE ROWS]
```

- Use the SERIAL data type to specify a NOT NULL and auto-incrementing INTEGER column. The starting value of a SERIAL column is 1.

Implicit creation syntax

Local temporary tables can be defined implicitly by referencing them in an INSERT statement.

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

Note	If `#name` does not exist, it is defined using the given column names and types from the value expressions.

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

Note	If `#name` does not exist, it is defined using the target column names, and the types from the query-derived columns. If target columns are not supplied, the column names will match the derived column names from the query.

Drop syntax

```
DROP TABLE name
```

+ In the following example, a series of statements loads a temporary table with data from 2 sources, manually inserts a record, and then uses the temporary table in a SELECT query.

Example: Local temporary tables

```
CREATE LOCAL TEMPORARY TABLE TEMP (a integer, b integer, c integer);
SELECT * INTO temp FROM Src1;
SELECT * INTO temp FROM Src2;
INSERT INTO temp VALUES (1,2,3);
SELECT a,b,c FROM Src3, temp WHERE Src3.a = temp.b;
```

For more information about using local temporary tables, see Virtual procedures.

Global temporary tables

Global temporary tables are created from the metadata that you supply to Teiid at deployment time. Unlike local temporary tables, you cannot create global temporary tables at runtime. Your global temporary tables share a common definition through a schema entry. However, a new instance of the temporary table is created in each session. The table is then dropped when the session ends. There is no explicit drop support. A common use for a global temporary table is to pass results into and out of procedures.

Creation syntax

```
CREATE GLOBAL TEMPORARY TABLE name (column type [NOT NULL], ... [PRIMARY KEY (column, ...)]) OPTIONS (UPDATABLE
'true')
```

If you use the SERIAL data type, then each session's instance of the global temporary table will have its own sequence.

You must explicitly specify UPDATABLE if you want to update the temporary table.

For information about syntax options, see the `CREATE TABLE` DDL statements in DDL metadata for schema objects.

Common features of global and local temporary tables

Global and local temporary tables share some common features.

Primary key usage

- All key columns must be comparable.
- If you use a primary key, it creates a clustered index that enables search improvements for SQL comparison operators, and the IN, LIKE, and ORDER BY operators.
- You can use `Null` as a primary key value, but there must be only one row that has an all-null key.

Transactions

- There is a `READ_UNCOMMITED` transaction isolation level. There are no locking mechanisms available to enable higher isolation levels, and the result of a rollback may be inconsistent across multiple transactions. If concurrent transactions are not associated with the same local temporary table or session, then the transaction isolation level is effectively serializable. If you want full consistency with local temporary tables, then only use a connection with one transaction at a time. This mode of operation is ensured by connection pooling that tracks connections by transaction.

Limitations

- With the `CREATE TABLE` syntax, you can specify only basic table definition (column name, type, and nullable information), and an optional primary key. For global temporary tables, additional metadata in the CREATE statement is effectively ignored when creating the temporary table instance. However, the metadata might still be used by planning similar to any other table entry.
- You can use `ON COMMIT PRESERVE ROWS`. You cannot use other `ON COMMIT` actions.
- The cannot use "drop behavior" options in the DROP statement.
- Temporary tables are not fail-over safe.
- Non-inlined LOB values (XML, CLOB, BLOB, JSON, geometry) are tracked by reference rather than by value in a temporary table. If you insert LOB values from external sources in your temporary table, they might become unreadable when the associated statement or connection is closed.

Foreign temporary tables

Unlike a local or global temporary table, a foreign temporary table is a reference to an actual source table that is created at runtime, rather than during the metadata load.

A foreign temporary table requires explicit creation syntax:

```
CREATE FOREIGN TEMPORARY TABLE name ... ON schema
```

Where the table creation body syntax is the same as a standard CREATE FOREIGN TABLE DDL statement. For more information, see DDL metadata. In general, usage of DDL OPTION clauses might be required to properly access the source table, including setting the name in the source, updatability, native types, and so forth.

The schema name must specify an existing schema/model in the VDB. The table will be accessed as if it is on that source. However within 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 it is created.

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

Neither a CREATE nor a corresponding DROP of a foreign temporary table issues a pushdown command. Rather, this mechanism exposes a source table for use within 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 on the CREATE statement. Some other mechanism, such as the native procedure shown above, must be used to first create the table. Also note the table is explicitly marked as updatable, since DDL defined tables are not updatable by default.

The source's handling of temporary tables must also be understood to make this work as intended. Sources that use the same GLOBAL table definition for all sessions while scoping the data to be session-specific (such as Oracle) or sources that use session-scoped temporary tables (such as PostgreSQL) will work if accessed under a transaction. A transaction is necessary for the following reasons:

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

Tip	You cannot use the `ON COMMIT` clause with Teiid. As a result, for local temporary tables, the `ON COMMIT` behavior for source tables is likely to be different from the default `PRESERVE ROWS` .

Alter view

Usage

```
ALTER VIEW name AS queryExpression
```

Syntax rules

- The alter query expression can be prefixed with a cache hint for materialized view definitions. The hint takes effect the next time that the materialized view table loads.

Alter procedure

Usage

```
ALTER PROCEDURE name AS block
```

Syntax rules

- The ALTER block should not include `CREATE VIRTUAL PROCEDURE`.
- You can prefix the ALTER block with a cache hint for cached procedures.

Alter trigger

Usage

```
ALTER TRIGGER ON name INSTEAD OF INSERT|UPDATE|DELETE (AS FOR EACH ROW block) | (ENABLED|DISABLED)
```

Syntax rules

- The target `name` must be an updatable view.
- Triggers are not true schema objects. They are scoped only to their view and have no name.
- Update procedures must already exist for the given trigger event. For more information, see Triggers.

Procedures

You can use a procedure language in Teiid to call foreign procedures and define virtual procedures and triggers.

Procedure language

You can use a procedural language in Teiid to define virtual procedures. These are similar to stored procedures in relational database management systems. You can use this language to define the transformation logic for decomposing INSERT, UPDATE, and DELETE commands against views. These are known as update procedures. For more information, see Virtual procedures and update procedures (Triggers).

Command statement

A command statement executes a DML command, DDL command, or dynamic SQL against one or more data sources. For more information, see DML commands and DDL commands.

Usage

```
command [(WITH|WITHOUT) RETURN];
```

Example command statements

```
SELECT * FROM MySchema.MyTable WHERE ColA > 100 WITHOUT RETURN;
INSERT INTO MySchema.MyTable (ColA,ColB) VALUES (50, 'hi');
```

Syntax rules

- EXECUTE command statements may access IN/OUT, OUT, and RETURN parameters. To access the return value the statement will have the form `var = EXEC proc…`. To access OUT or IN/OUT values named parameter syntax must be used. For example, `EXEC proc(in_param⇒'1', out_param⇒var)` will assign the value of the out parameter to the variable var. It is expected that the datatype of a parameter is implicitly convertible to the data type of the variable. For more information about EXECUTE command statements, see EXECUTE command.
- The RETURN clause determines if the result of the command is returnable from the procedure. WITH RETURN is the default. If the command does not return a result set, or the procedure does not return a result set, the RETURN clause is ignored. If WITH RETURN is specified, the result set of the command must match the expected result set of the procedure. Only the last successfully executed statement executed WITH RETURN will be returned as the procedure result set. If there are no returnable result sets and the procedure declares that a result set will be returned, then an empty result set is returned.

Note

The INTO clause is used only for inserting into a table. `SELECT … INTO table … ` is functionally equivalent to `INSERT INTO table SELECT … ` If you need to assign variables, you can use one of the following methods:

Use an assignment statement with a scalar subquery

```
DECLARE string var = (SELECT col ...);
```

Use a temporary table

```
INSERT INTO #temp SELECT col1, col2 ...;
DECLARE string VARIABLES.RESULT = (SELECT x FROM #temp);
```

Use an array

```
DECLARE string[] var = (SELECT (col1, col2) ...);
DECLARE string col1val = var[1];
```

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 or string value of less than 262144 characters.
- The `AS` clause is used to define the projected symbols names and types returned by the executed SQL string. The `AS` clause symbols will be matched positionally with the symbols returned by the executed SQL string. Non-convertible types or too few columns returned by the executed SQL string will result in an error.
- The `INTO` clause will project the dynamic SQL into the specified temp table. With the `INTO` clause specified, the dynamic command will actually execute a statement that behaves like an INSERT with a QUERY EXPRESSION. If the dynamic SQL command creates a temporary table with the `INTO` clause, then the `AS` clause is required to define the table's metadata. Note that if the temporary table already exists, then the insert columns are matched positionally - not by name.
- The `USING` clause allows the dynamic SQL string to contain variable references that are bound at runtime to specified values. This allows for some independence of the SQL string from the surrounding procedure variable names and input names. In the dynamic command `USING` clause, each variable is specified by short name only. However, in the dynamic SQL the `USING` variable must be fully qualified to `DVAR`. The `USING` clause is only for values that will be used in the dynamic SQL as valid expressions. It is not possible to use the `USING` clause to replace table names, keywords, and so forth. This makes using symbols equivalent in power to normal bind (?) expressions in prepared statements. The `USING` clause helps reduce the amount of string manipulation needed. If a reference is made to a USING symbol in the SQL string that is not bound to a value in the `USING` clause, an exception will occur.
- The `UPDATE` clause is used to specify the updating model count. Accepted values are (0,1,*). 0 is the default value if the clause is not specified. For more information, see Updating model count.

Example: Dynamic SQL

```
...
/* Typically complex criteria would be formed based upon inputs to the procedure.
 In this simple example the criteria is references the using clause to isolate
 the SQL string from referencing a value from the procedure directly */

DECLARE string criteria = 'Customer.Accounts.Last = DVARS.LastName';

/* Now we create the desired SQL string */
DECLARE string sql_string = 'SELECT ID, First || " " || Last AS Name, Birthdate FROM Customer.Accounts WHERE '
|| criteria;

/* The execution of the SQL string will create the #temp table with the columns (ID, Name, Birthdate).
  Note that we also have the USING clause to bind a value to LastName, which is referenced in the criteria. */
EXECUTE IMMEDIATE sql_string AS ID integer, Name string, Birthdate date INTO #temp USING LastName='some name';

/* The temp table can now be used with the values from the Dynamic SQL */
loop on (SELCT ID from #temp) as myCursor
...
```

Here is an example showing a more complex approach to building criteria for the dynamic SQL string. In short, the virtual procedure `AccountAccess.GetAccounts` has the inputs `ID`, `LastName`, and `bday`. If a value is specified for `ID` it will be the only value used in the dynamic SQL criteria. Otherwise, if a value is specified for `LastName` the procedure will detect if the value is a search string. If `bday` is specified in addition to `LastName`, it will be used to form compound criteria with `LastName`.

Example: Dynamic SQL with USING clause and dynamically built criteria string

```
...
DECLARE string crit = null;

IF (AccountAccess.GetAccounts.ID IS NOT NULL)
 crit = '(Customer.Accounts.ID = DVARS.ID)';
ELSE IF (AccountAccess.GetAccounts.LastName IS NOT NULL)
BEGIN
 IF (AccountAccess.GetAccounts.LastName == '%')
   ERROR "Last name cannot be %";
 ELSE IF (LOCATE('%', AccountAccess.GetAccounts.LastName) < 0)
   crit = '(Customer.Accounts.Last = DVARS.LastName)';
 ELSE
   crit = '(Customer.Accounts.Last LIKE DVARS.LastName)';
 IF (AccountAccess.GetAccounts.bday IS NOT NULL)
   crit = '(' || crit || ' and (Customer.Accounts.Birthdate = DVARS.BirthDay))';
END
ELSE
 ERROR "ID or LastName must be specified.";

EXECUTE IMMEDIATE 'SELECT ID, First || " " || Last AS Name, Birthdate FROM Customer.Accounts WHERE ' || crit US
ING ID=AccountAccess.GetAccounts.ID, LastName=AccountAccess.GetAccounts.LastName, BirthDay=AccountAccess.GetAcc
ounts.Bday;
...
```

Dynamic SQL limitations and workarounds
The use of the dynamic SQL command results in an assignment statement that requires the use of a temporary table.

Example assignment

```
EXECUTE IMMEDIATE <expression> AS x string INTO #temp;
DECLARE string VARIABLES.RESULT = (SELECT x FROM #temp);
```

The construction of appropriate criteria will be cumbersome if parts of the criteria are not present. For example if `criteria` were already NULL, then the following example results in `criteria` remaining NULL.

Example: Dangerous NULL handling

```
...
criteria = '(' || criteria || ' and (Customer.Accounts.Birthdate = DVARS.BirthDay))';
```

It is best to ensure that the criteria is not NULL prior its usage. If this is not possible, a you can specify a default, as shown in the following example.

Example: NULL handling

```
...
criteria = '(' || nvl(criteria, '(1 = 1)') || ' and (Customer.Accounts.Birthdate = DVARS.BirthDay))';
```

If the dynamic SQL is an `UPDATE`, `DELETE`, or `INSERT` command, the rowcount of the statement can be obtained from the rowcount variable.

Example: AS and INTO clauses

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

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 by their fully qualified names.

Example: Out parameter

```
CREATE VIRTUAL PROCEDURE proc (OUT STRING x, INOUT STRING y) AS
BEGIN
  proc.x = 'some value ' || proc.y;
  y = 'some new value';
END
```

Special variables

`VARIABLES.ROWCOUNT` integer variable will contain the numbers of rows affected by the last INSERT, UPDATE, or DELETE command statement executed. Inserts that are processed by dynamic SQL with an `into` clause will also update the `ROWCOUNT` .

Sample usage

```
...
UPDATE FOO SET X = 1 WHERE Y = 2;
DECLARE INTEGER UPDATED = VARIABLES.ROWCOUNT;
...
```

Non-update command statements (`WITH` or `WITHOUT RETURN`) will reset the `ROWCOUNT` to 0.

Note	To ensure you are getting the appropriate `ROWCOUNT` value, save the `ROWCOUNT` to a variable immediately after the command statement.

Compound statement

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

Usage

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

Note	When a block is expected by an `IF` , `LOOP` , `WHILE` , and so forth, a single statement is also accepted by the parser. Even though the block `BEGIN` or `END` are not expected, the statement will execute as if wrapped in a `BEGIN` or `END` pair.

Syntax rules

- If `NOT ATOMIC` or no `ATOMIC` clause is specified, the block will be executed non-atomically.
- If the `ATOMIC` clause is specified, the block must execute atomically. If a transaction is already associated with the thread, no additional action will be taken; savepoints or sub-transactions are not currently used. If the higher level transaction is used, and the block does not complete — regardless of the presence of exception handling — the transaction will be marked as rollback only. Otherwise, a transaction will be associated with the execution of the block. Upon successful completion of the block the transaction will be committed.
- The label must not be the same as any label that is used in statements that contain this one.
- Variable assignments and the implicit result cursor are unaffected by rollbacks. If a block does not complete successfully, its assignments will still take affect.

Exception handling

If an `EXCEPTION` clause is used within a compound statement, any processing exception emitted from statements will be caught with the flow of execution transferring to `EXCEPTION` statements. Any block-level transaction started by this block will commit if the exception handler successfully completes. If another exception, or the original exception, is emitted from the exception handler, the transaction will rollback. Any temporary tables or variables specific to the BLOCK will not be available to the exception handler statements.

Note	Only processing exceptions, which are typically caused by errors originating at the sources or with function execution, are caught. A low-level internal 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 following table shows the variables that an exception group contains:

Variable	Type	Description
STATE	string	The SQL State
ERRORCODE	integer	The error or vendor code. In the case of 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 might not be the same as any higher level exception group or loop cursor name.

Example exception group handling

```
BEGIN
    DECLARE EXCEPTION e = SQLEXCEPTION 'this is bad' SQLSTATE 'xxxxx';
    RAISE variables.e;
EXCEPTION e
    IF (e.state = 'xxxxx')
        //in this trivial example, we'll always hit this branch and just log the exception
        RAISE SQLWARNING e.exception;
    ELSE
        RAISE e.exception;
END
```

IF statement

An IF statement evaluates a condition and executes either one of two statements depending on the result. You can nest IF statements to create complex branching logic. A dependent ELSE statement will execute its statement only if the IF statement evaluates to `false`.

Usage

```
IF (criteria)
   block
[ELSE
   block]
END
```

Example IF statement

```
IF ( var1 = 'North America')
BEGIN
  ...statement...
END ELSE
BEGIN
  ...statement...
END
```

The criteria can be any valid Boolean expression or an `IS DISTINCT FROM` predicate referencing row values. The `IS DISTINCT FROM` extension uses the following syntax:

```
rowVal IS [NOT] DISTINCT FROM rowValOther
```

Where `rowVal` and `rowValOther` are references to row value group. This would typically be used in instead of update triggers on views to quickly determine if the row values are changing:

Example: IS DISTINCT FROM IF statement

```
IF ( "new" IS DISTINCT FROM "old")
BEGIN
  ...statement...
END
```

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

Tip	Null values should be considered in the criteria of an IF statement. `IS NULL` criteria can be used to detect the presence of a null value.

Loop Statement

A LOOP statement is an iterative control construct that is used to cursor through a result set.

Usage

```
[label :] LOOP ON <select statement> AS <cursorname>
    statement
```

Syntax rules

- The label must not be the same as any label that is used in statements that contain this one.

While statement

A `WHILE` statement is an iterative control construct that is used to execute a statement repeatedly whenever a specified condition is met.

Usage

```
[label :] WHILE <criteria>
    statement
```

Syntax rules

- The label must not be the same as any label that is used in statements that contain this one.

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 convertible to the expected type.
- Even if the procedure has a return parameter, it is not required to specify a return value in a `RETURN` statement. A return parameter can be set through an assignment or it can be left as null.

Sample usage

```
CREATE VIRTUAL FUNCTION times_two(val integer)
   RETURNS integer AS
   BEGIN
      RETURN val*2;
   END
```

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 that specify the exception message and SQL state. Teiid does not fully comply with the ANSI SQL specification on SQL state usage, but you are allowed to set any SQL state you choose.
- `code` is an integer expression that specifies the vendor code.
- `exception` must be a variable reference to an exception or an exception expression, and will be chained to the resulting exception as its parent.

Virtual procedures

Virtual procedures are defined using the Teiid procedural language. For more information, see Procedure language.

A virtual procedure has zero or more INPUT, INOUT, or OUT parameters, an optional RETURN parameter, and an optional result set. Virtual procedures can execute queries and other SQL commands, define temporary tables, add data to temporary tables, walk through result sets, use loops, and use conditional logic.

Virtual procedure definition
For more information, see *Create procedure/function* in DDL metadata for schema objects.

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

Within the body of the procedure, you can use any valid statement. For more information avbout procedure language statements, see Procedure language.

There is no explicit cursoring or value returning statement. Instead, the last unnamed command statement executed in the procedure that returns a result set will be returned as the result. The output of that statement must match the expected result set and parameters of the procedure.

Virtual procedure parameters
Virtual procedures can take zero or more `IN` or `INOUT` parameters, and can have any number of `OUT` parameters and an optional `RETURN` parameter. Each input has the following information that is used during runtime processing:

Name
: The name of the input parameter.

Datatype
: The design-time type of the input parameter.

Default value
: The default value if the input parameter is not specified.

Nullable
: `NO_NULLS` , `NULLABLE` , `NULLABLE_UNKNOWN` ; parameter is optional if nullable, and is not required to be listed when using named parameter syntax.

You reference a parameter in a virtual procedure by using its fully-qualified name (or less if unambiguous). For example, `MySchema.MyProc.Param1` .

Example: Referencing an input parameter and assigning an Out parameter for `GetBalance` procedure

```
BEGIN
  MySchema.GetBalance.RetVal = UPPER(MySchema.GetBalance.AcctID);
  SELECT Balance FROM MySchema.Accts WHERE MySchema.Accts.AccountID = MySchema.GetBalance.AcctID;
END
```

If an `INOUT` parameter is not assigned any value in a procedure, it will retain the value it was assigned for input. Any `OUT/RETURN` parameter that is not assigned a value will retain the default NULL value. The `INOUT/OUT/RETURN` output values are validated against the `NOT NULL` metadata of the parameter.

Example virtual procedures
The following example represents a loop that walks through a cursored table and uses `CONTINUE` and `BREAK` .

Virtual procedure using LOOP, CONTINUE, BREAK

```
BEGIN
  DECLARE double total;
```

```
  DECLARE integer transactions;
  LOOP ON (SELECT amt, type FROM CashTxnTable) AS txncursor
  BEGIN
    IF(txncursor.type <> 'Sale')
    BEGIN
      CONTINUE;
    END ELSE
    BEGIN
      total = (total + txncursor.amt);
      transactions = (transactions + 1);
      IF(transactions = 100)
      BEGIN
        BREAK;
      END
    END
  END
  SELECT total, (total / transactions) AS avg_transaction;
END
```

The following example uses conditional logic to determine which of two SELECT statements to execute.

Virtual procedure with conditional SELECT

```
BEGIN
  DECLARE string VARIABLES.SORTDIRECTION;
  VARIABLES.SORTDIRECTION = PartsVirtual.OrderedQtyProc.SORTMODE;
  IF ( ucase(VARIABLES.SORTDIRECTION) = 'ASC' )
  BEGIN
    SELECT * FROM PartsVirtual.SupplierInfo WHERE QUANTITY > PartsVirtual.OrderedQtyProc.QTYIN ORDER BY 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. For more information, see *Execute command* in DML commands.

If the procedure has defined inputs, you specify those in a sequential list, or using *name=value* syntax. You must use the name of the input parameter, scoped by the full procedure name if the parameter name is ambiguous in the context of other columns or variables in the procedure.

A virtual procedure call returns a result set like any `SELECT`, so you can use this in many places you can use a `SELECT`. Typically you'll use the following syntax:

```
SELECT * FROM (EXEC ...) AS x
```

Virtual procedure limitations

A virtual procedure can return only one result set. If you need to pass in a result set, or pass out multiple result sets, then consider using global temporary tables instead.

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 that you run against a view (`INSERT` , `UPDATE` , or `DELETE`) require logic to define how the tables and views integrated by the view are affected by each type of command. This transformation logic, also referred to as a **trigger**, is invoked when an update command is issued against a view. Update procedures define the logic for how the update command that you run against a view is decomposed into the individual commands to be executed against the underlying physical sources. Similar to virtual procedures, update procedures have the ability to execute queries or other commands, define temporary tables, add data to temporary tables, walk through result sets, use loops, and use conditional logic. For more inmformation about virtual procedures, see Virtual procedures.

You can use `INSTEAD OF` triggers on views in a way that is similar to the way that you might use them with traditional databases. You can have only one `FOR EACH ROW` procedure for each `INSERT` , `UPDATE` , or `DELETE` operation against a view.

Usage

```
CREATE TRIGGER ON view_name INSTEAD OF INSERT|UPDATE|DELETE AS
FOR EACH ROW
...
```

Update procedure processing

1. The user application submits the SQL command.
2. The command detects the view that it is executed against.
3. The correct procedure is chosen depending upon the command type (`INSERT` , `UPDATE` , or `DELETE`).
4. The procedure is executed. The procedure might contain SQL commands of its own. Commands in the procedure can be different in type from the command that is received from the calling application.
5. Commands, as described in the procedure, are issued to the individual physical data sources or other views.
6. A value representing the number of rows changed is returned to the calling application.

Source triggers

Teiid can use `AFTER` triggers on source tables. `AFTER` triggers are called by events from a change data capture (CDC) system.

Usage:

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

FOR EACH ROW triggers

Only the `FOR EACH ROW` construct serves as a trigger handler. A `FOR EACH ROW` trigger procedure will evaluate its block for each row of the view/source affected by the `UPDATE` statement. For `UPDATE` and `DELETE` statements, this will be every row that passes the `WHERE` condition. For `INSERT` statements there will be one new row for each set of values from the `VALUES` or query expression. For a view, the rows updated is reported as this number, regardless of the affect of the underlying procedure logic.

Usage

```
FOR EACH ROW
   BEGIN ATOMIC
      ...
   END
```

The `BEGIN` and `END` keywords are used to denote block boundaries. Within the body of the procedure, any valid statement may be used.

Note	The use of the `ATOMIC` keyword is currently optional for backward compatibility, but unlike a normal block, the default for `INSTEAD OF` triggers is atomic.

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

NEW variables

Every attribute in the view/table whose `UPDATE` and `INSERT` transformations you are defining has an equivalent variable named `NEW.<column_name>`.

When an INSERT or an UPDATE command is executed against the view, or the event is received, these variables are initialized to the values in the `INSERT VALUES` clause or the `UPDATE SET` clause respectively.

In an `UPDATE` procedure, the default value of these variables, if they are not set by the command, is the old value. In an INSERT procedure, the default value of these variables is the default value of the virtual table attributes. See `CHANGING` variables, later in this list for distinguishing defaults from passed values.

OLD variables

Every attribute on the view/table whose `UPDATE` and `DELETE` transformations you are defining has an equivalent variable named `OLD.<column_name>`.

When a `DELETE` or `UPDATE` command is executed against the view, or the event is received, these variables are initialized to the current values of the row being deleted or updated respectively.

CHANGING variables

Every attribute on the view/table whose `UPDATE` and `INSERT` transformations you are defining has an equivalent variable named `CHANGING.<column_name>`.

When an `INSERT` or an `UPDATE` command is executed against the view, or an the event is received, these variables are initialized to `true` or `false` depending on whether the `INPUT` variable was set by the command. A `CHANGING` variable is commonly used to differentiate between a default insert value and one that is specified in the user query.

For example, for a view with columns A, B, C:

If User Executes…	Then…
`INSERT INTO VT (A, B) VALUES (0, 1)`	CHANGING.A = true, CHANGING.B = true, CHANGING.C = false
`UPDATE VT SET C = 2`	CHANGING.A = false, CHANGING.B = false, CHANGING.C = true

Key variables

To return generated keys from an `INSERT` trigger, a KEY group is made available that can be assigned the value to be returned. Typically this requires using the `generated_key` system function. However, not all inserts provide generated keys, because not all sources return generated keys.

```
create view v1 (i integer, k integer not null auto_increment primary key)
OPTIONS (UPDATABLE true) as
   select x, y from tbl;
create trigger on v1 instead of insert as
   for each row begin atomic
      -- ... some logic
      insert into tbl (x) values (new.i);
      key.k = cast(generated_key('y') as integer);
   end;
```

Example update procedures

For example, for a view with columns A, B, C:

Sample DELETE procedure

```
FOR EACH ROW
BEGIN
    DELETE FROM X WHERE Y = OLD.A;
    DELETE FROM Z WHERE Y = OLD.A; // cascade the delete
END
```

Sample UPDATE procedure

```
FOR EACH ROW
BEGIN
    IF (CHANGING.B)
    BEGIN
        UPDATE Z SET Y = NEW.B WHERE Y = OLD.B;
    END
END
```

Other usages

`FOR EACH ROW` update procedures in a view can also be used to emulate `BEFORE/AFTER` each row triggers while still retaining the ability to perform an inherent update. This `BEFORE/AFTER` trigger behavior with an inherent update can be achieved by creating an additional updatable view over the target view with update procedures of the form:

```
CREATE TRIGGER ON outerVW INSTEAD OF INSERT AS
FOR EACH ROW
    BEGIN ATOMIC
    --before row logic
    ...

    --default insert/update/delete against the target view
    INSERT INTO VW (c1, c2, c3) VALUES (NEW.c1, NEW.c2, NEW.c3);

    --after row logic
    ...
    END
```

Comments

You can add multi-line SQL comments in Teiid by enclosing text with `/* */`.

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

You can also add single line comments:

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

You can also nest comments.

Explain statements

You can use an EXPLAIN statement to obtain a query plan. Using EXPLAIN statements to obtain a query execution plan is a native function of the SQL language, and it is the preferred mechanism to use over pg/ODBC transport. If you are using a Teiid JDBC client, you can also use SET/SHOW statements. For more information about SET and SHOW statements, see the Client Developer's Guide.

Usage

```
EXPLAIN [(explainOption [, ...])] statement

explainOption :=
      ANALYZE [TRUE|FALSE]
    | FORMAT {TEXT|YAML|XML}
```

If no options are specified, by default the plan is provided in text format without executing the query.

If you specify `ANALYZE` or `ANALYZE TRUE`, then the statement is executed, unless the client has set the `NOEXEC` option. The resulting plan will include runtime node statistics from the fully executed statement. All side effects, including updates, will still occur. You might need to use a transaction to rollback any unwanted side effects.

While this is superficially the same syntax as PostgreSQL, the plan provided in the various formats is the same that has been provided by Teiid in prior versions.

For more information about how to interpret results, see Query plans.

Example

```
EXPLAIN (analyze) select * from really_complicated_view
```

Returns a text-formatted plan from an actual run of the given statement.

Data types

The Teiid type system is based on Java/JDBC types. The runtime object is represented by the corresponding Java class, such as Long, Integer, Boolean, String, and so forth. For more information, see Runtime types. You can use domain types to extend the type system. For more information, see DDL metadata for domains.

Runtime types

Teiid works with a core set of runtime types. Runtime types can be different from semantic types that are defined in type fields at design time. The runtime type can also be specified at design time or it will be automatically chosen as the closest base type to the semantic type.

Note	Even if a type is declared with a length, precision, or scale argument, those restrictions are effectively ignored by the runtime system, but may be enforced/reported at the edge by OData, ODBC, JDBC. Geospatial types act in a similar manner. Extension metadata might be needed for SRID, type, and number of dimensions for consumption by tools/OData, but it is not yet enforced. In some instances you might need to use the ST_SETSRID function to ensure the SRID is associated.

Table 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 16 bit character - which cannot represent a value beyond the Basic Multilingual Plane. This limitation also applies to functions/expressions that expect a single character such as trim, textagg, texttable, and like escape.	java.lang.Character	CHAR	CHAR
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)	java.sql.Time	TIME	TIME
timestamp	Datetime, representing a single date and time (year, month, day, hours, minutes, seconds, fractional seconds).	java.sql.Timestamp	TIMESTAMP	TIMESTAMP
object	Any arbitrary Java object, must implement java.lang.Serializable.	Any	JAVA_OBJECT	VARCHAR
blob	Binary large object, representing a stream of bytes.	java.sql.Blob [2]	BLOB	VARCHAR
clob	Character large object, representing a stream of characters.	java.sql.Clob [3]	CLOB	VARCHAR
xml	XML document	java.sql.SQLXML[4]	JAVA_OBJECT	VARCHAR
geometry	Geospatial Object	java.sql.Blob [5]	BLOB	BLOB
geography (11.2+)	Geospatial Object	java.sql.Blob [6]	BLOB	BLOB

json (11.2+)	Character large object, representing a stream of JSON characters.	java.sql.Clob [7]	CLOB	VARCHAR

1. The runtime type is org.teiid.core.types.BinaryType. Translators will need to explicitly handle BinaryType values. UDFs will instead have a byte[] value passed.
2. The concrete type is expected to be org.teiid.core.types.BlobType
3. The concrete type is expected to be org.teiid.core.types.ClobType
4. The concrete type is expected to be org.teiid.core.types.XMLType
5. The concrete type is expected to be org.teiid.core.types.GeometryType
6. The concrete type is expected to be org.teiid.core.types.GeographyType
7. The concrete type is expected to be org.teiid.core.types.JsonType

Note	Character, String, and character large objects (CLOB) types are not limited to ASCII/extended ASCII values. Character can hold codes up to 2^16-1 and String/CLOB can hold any value.

Arrays

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

Example: Array types

```
string[]
```

```
integer[][]
```

Note	Array handling is typically in memory. It is not advisable to rely on the usage of large array values. Arrays of large objects (LOBs) are typically not handled correctly when serialized.

Type conversions

Data types may be converted from one form to another either explicitly or implicitly. Implicit conversions automatically occur in criteria and expressions to ease development. Explicit datatype conversions require the use of the `CONVERT` function or `CAST` keyword.

Type conversion considerations

- Any type may be implicitly converted to the `OBJECT` type.
- The `OBJECT` type can be explicitly converted to any other type.
- The NULL value can be converted to any type.
- Any valid implicit conversion is also a valid explicit conversion.
- In scenarios where literal values would normally require explicit conversions, you can apply implicit conversions if no loss of information occurs.
- If `widenComparisonToString` is false (the default), Teiid raises an exception if it detects that an explicit conversion cannot be applied implicitly in criteria.
- If `widenComparisonToString` is true, then depending upon the comparison, a widening conversion is applied or the criteria are treated as false. For more information about `widenComparisonToString`, see *System properties* in the Administrator's Guide.

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

  If `widenComparisonToString` is false, and `created_by` is a date, `not a date` cannot be converted to a date value, and an exception results.
- Explicit conversions that are not allowed between two types will result in an exception before execution. Allowed explicit conversions can still fail during processing if the runtime values are not actually convertible.

Warning	The 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 can vary depending upon the actual source type and conversion logic. It is best not to assume use of the string form in criteria or other places where variations might lead to different results.

Table 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). For more information, see *XMLPARSE* in XML functions.
2. Implicit conversion to float/double only occurs for literal values.
3. xml to string is equivalent to XMLSERIALIZE(exp AS STRING). For more information, see *XMLSERIALIZE* in XML functions.

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. For example:

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

In the preceding example, if the `created_by` column has the data type of date, Teiid automatically converts the data type of the string literal to a date.

Converting to Boolean
Teiid can automatically convert literal strings and numeric type values to Boolean values as shwon in the following table:

Table 1. Boolean conversions

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

Date and time conversions
Teiid can implicitly convert properly formatted literal strings to their associated date-related data types as shown in the following table:

Table 2. Date and time conversions

String literal format	Possible implicit conversion type
yyyy-mm-dd	DATE
hh:mm:ss	TIME
yyyy-mm-dd[hh:mm:ss.[fff…]]	TIMESTAMP

The preceding formats are those expected by the JDBC date types. For information about using other formats, see the functions `PARSEDATE` , `PARSETIME` , and `PARSETIMESTAMP` in Date and time functions.

Escaped literal syntax

Rather than relying on implicit conversion, you can define data type values directly in SQL by using escape syntax. The string values that you supply must match the expected format exactly, or an exception will occur.

Datatype	Escaped syntax	Standard literal
BOOLEAN	{b 'true'}	TRUE
DATE	{d 'yyyy-mm-dd'}	DATE 'yyyy-mm-dd'
TIME	{t 'hh-mm-ss'}	TIME 'hh-mm-ss'
TIMESTAMP	{ts 'yyyy-mm-dd[hh:mm:ss.[fff…]]'}	TIMESTAMP 'yyyy-mm-dd[hh:mm:ss.[fff…]]'

Updatable Views

Any view can be marked as *updatable*. In many circumstances the view definition allows the view to be inherently updatable without the need to manually define a trigger to handle `INSERT/UPDATE/DELETE` operations.

An inherently updatable view cannot be defined with a query that has:

- A set operation (`INTERSECT` , `EXCEPT` , `UNION`).
- `SELECT DISTINCT` .
- Aggregation (aggregate functions, `GROUP BY` , `HAVING`).
- A `LIMIT` clause.

A `UNION ALL` can define an inherently updatable view only if each of the UNION branches are themselves inherently updatable. A view defined by a `UNION ALL` can accommodate inherent `INSERT` statements if it is a partitioned union, and the `INSERT` specifies values that belong to a single partition. For more information, see *partitioned union* in Federated optimizations.

Any view column that is not mapped directly to a column is not updatable and cannot be targeted by an UPDATE set clause or be an INSERT column.

If a view is defined by a join query or has a `WITH` clause it might still be inherently updatable. However, in these situations there are further restrictions, and the resulting query plan may execute multiple statements. For a non-simple query to be updatable, the following criteria apply:

- An `INSERT/UPDATE` can only modify a single key-preserved table.
- To allow `DELETE` operations, there must be only a single key-preserved table.

  For information about key-preserved tables, see Key-preserved tables.

If the default handling is not available or if you want to have an alternative implementation of an `INSERT/UPDATE/DELETE` , you can use update procedures, or triggers, to define procedures to handle the respective operations. For more information see Update procedures (Triggers).

Consider the following example of an inherently updatable denormalized view:

```
create foreign table parent_table (pk_col integer primary key, name string) options (updatable true);

create foreign table child_table (pk_col integer primary key, name string, fk_col integer, foreign key (fk_col)
references parent_table (pk_col)) options (updatable true);

create view denormalized options (updatable true) as select c.fk_col, c.name as child_name, p.name from parent_
table as p, child_table as c where p.pk_col = c.fk_col;
```

A query such as `insert into denormalized (fk_col, child_name) values (1, 'a')` would succeed against this view, because it targets a single key-preserved table, `child_table` . However, `insert into denormalized (name) values ('a')` would fail, because it maps to a `parent_table` that can have multiple rows for each `parent_table` key. In other words, it is not *key-preserved*.

Also, an `INSERT` against `parent_table` alone might not be visible to the view, because there might be no child entities associated either.

Not all scenarios will work. Referencing the preceding example, an `insert into denormalized (pk_col, child_name) values (1, 'a')` with a view that is defined using the `p.pk_col` will fail, because the logic doesn't yet consider the equivalency of the key values.

Transactions

Teiid utilizes XA transactions for participating in global transactions and for demarcating its local and command scoped transactions.

Narayana is used by Teiid as its transaction manager.

Narayana is optionally used by Teiid as its transaction manager. In single source, or scenarios utilizing only non-XA sources, then the Spring platform transaction manager may be used.

For information about advanced transaction technologies that are provided for Teiid through the Narayana community project, see the Narayana documentation.

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

User level commands can execute multiple source commands. To control the transactional behavior of a user command when not in a local or global transaction, you can specify the AutoCommitTxn execution property.

Table 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 the following criteria are true:

- The user command is fully pushed to the source.
- The user command is a SELECT (including XML) and the transaction isolation is not REPEATABLE_READ nor SERIALIABLE.
- The user command is a stored procedure, the transaction isolation is not REPEATABLE_READ nor SERIALIABLE, and the updating model count is zero. For more information, see Updating model count.

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

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). The success or failure of that update corresponds to the success or failure of the command. It should not be possible for the update to succeed while the command fails. Execution is not considered transactionally unsafe.
*	Any number greater than 1 indicates that execution is transactionally unsafe and an XA transaction will be required.

JDBC and transactions

JDBC API functionality

The transaction scopes in Transactions map to the following JDBC modes:

Command

Connection autoCommit property set to true.

Local

Connection autoCommit property set to false. The transaction is committed by setting autoCommit to true or calling `java.sql.Connection.commit`. The transaction can be rolled back by a call to `java.sql.Connection.rollback`

Global

The XAResource interface provided by an XAConnection is used to control the transaction. Note that XAConnections are available only if 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 the following ways to manage transactions for beans:

Client-controlled

The client of a bean begins and ends a transaction explicitly.

Bean-managed

The bean itself begins and ends a transaction explicitly.

Container-managed

The application server container begins and ends a transaction automatically.

In any of the preceding cases, transactions can be either local or XA transactions, depending on how the code and descriptors are written. The XA specification does not require some types of beans (for example, stateful session beans and entity beans) to work with non-transactional sources. However, according to the specification, optionally, application servers can allow the use of these beans with non-transactional sources, with the caution that such usage is not portable or predictable. Generally speaking, to provide for most types of EJB activities in a portable fashion, applications require a mechanism for managing transactions.

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**

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

- The client setting of transaction isolation level is propagated only to JDBC connectors; the setting is not propagated to other connector types. The default transaction isolation level can be set on each XA connector. However, the isolation level is fixed, and cannot be changed at runtime for specific connections or commands.

Data roles

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

Before you apply data roles, you might want to restrict source system access through the fundamental design of your virtual database. 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 virtual database.

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

Warning	A virtual database that is deployed without data roles can be accessed by any authenticated user. 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.

Tip	By default, non-hidden schema metadata is only visible over JDBC/pg if the user is permissioned in some way for the given object. OData access provides all non-hidden metadata by default. To configure JDBC/pg to also make all non-hidden schema metadata visible to all authenticated users, set the environment/system property `org.teiid.metadataRequiresPermission` to false.

Permissions

Permissions, or grants, control access to data in several ways. There are simple access restrictions to SELECT, UPDATE, and so forth, down to a column level.

Note	Column or table metadata are not visible to JDBC/ODBC users unless the user has permission to read at least a single column.

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

User query permissions
CREATE, READ, UPDATE, DELETE (CRUD) permissions can be set for any resource path in a VDB. A resource path can be as specific as the fully qualified name of a column or as general a top level model (schema) name. Permissions granted to a particular path apply to it and any resource paths that share the same partial name. For example, granting select to "model" will also grant select to "model.table", "model.table.column", and so on. Allowing or denying a particular action is determined by searching for permissions from the most to least specific resource paths. The first permission found with a specific allow or deny will be used. Thus, it is possible to set very general permissions at high-level resource path names and to override only as necessary at more specific resource paths.

Permission grants are only needed for resources that a role needs access to. Permissions are also applied only to the columns/tables/procedures in the user query, not to every resource that is accessed transitively through view and procedure definitions. It is important therefore to ensure that permission grants are applied consistently across models that access the same resources.

Warning	Non-visible models are accessible by user queries. To restrict user access at a model level, at least one data role should be created to enable data role checking. In turn, that role can be mapped to any authenticated user, and should not grant permissions to models that should be inaccessible.

Permissions are not applicable to the SYS and pg_catalog schemas. These metadata reporting schemas are always accessible regardless of the user. The SYSADMIN schema however may need permissions as applicable.

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

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

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

- *INSERT*- on the Table being inserted into.
- *INSERT*- on every column being inserted on that Table.

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

- *UPDATE*- on the Table being updated.
- *UPDATE*- on every column being updated on that Table.
- *SELECT*- on every column referenced in the criteria.

To process a *DELETE* statement, the user account requires the following access rights:

- *DELETE*- on the Table being deleted.
- *SELECT*- on every column referenced in the criteria.

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

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

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

- *EXECUTE (or SELECT)*- on the Function being called.

To process any ALTER or CREATE TRIGGER statement, the user account requires the following access rights:

- *ALTER*- on the view or procedure that is effected. INSTEAD OF Triggers (update procedures) are not yet treated as full schema objects and are instead treated as attributes of the view.

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

- *LANGUAGE* - specifying the language name that is allowed.

To process any statement against a Teiid temporary table requires the following access rights:

- allow-create-temporary-tables attribute on any applicable role
- *SELECT*,*INSERT*,*UPDATE*,*DELETE* - against the target model/schema as needed for operations against a FOREIGN temporary table.

Row- and column-based security
Although specified in a similar way to user query CRUD permissions, row-based and column-based permissions may be used together or separately to control the data that is returned to users at a more granular and consistent level.

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

Note	Row-based security Specifying a condition on a GRANT for row based security has been deprecated. Specifying a condition on a GRANT is the same as specifying "CREATE POLICY policyName ON schemaName.tblName TO role USING (condition);", such that the condition applies to all operations.

A POLICY against a fully qualified table/view/procedure may specify a condition to be satisfied by the given role. The condition can be any valid boolean expression referencing the columns of the table/view/procedure. Procedure result set columns may be referenced as `proc.col`. The condition will act as a row-based filter and as a checked constraint for insert/update operations.

Application of row-based conditions
A condition is applied conjunctively to update/delete/select WHERE clauses against the affected resource. Those queries will therefore only ever be effective against the subset of rows that pass the condition, such as "SELECT * FROM TBL WHERE something **AND condition**. The condition will be present regardless of how the table/view is used in the query, whether by means of a union, join, or other operation.

Example condition

```
CREATE POLICY policyName ON schemaName.tblName TO superUser USING ('foo=bar');
```

Inserts and updates against physical tables affected by a condition are further validated so that the insert/change values must pass the condition (evaluate to true) for the insert/update to succeed — this is effectively the same a SQL constraint. This will happen for all styles of insert/update — insert with query expression, bulk insert/update, and so on. Inserts/updates against views are not checked with regards to the constraint.

You can disable the insert/update constraint check by restricting the operations that the POLICY applies to.

Example DDL non-constraint condition

```
CREATE POLICY readPolicyName ON schemaName.tblName FOR SELECT,DELETE TO superUser USING ('col>10');
```

You may of course add another POLICY to cover the INSERT and UPDATE operations should they require a different condition.

Example XML non-constraint condition

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

If more than one POLICY applies to the same resource, the conditions will be accumulated disjunctively via OR, that is, " (condition1) **OR** (condition2) …". Therefore, creating a POLICY with the condition "true" will allow users in that role to see all rows of the given resource for the given operations.

Considerations when using conditions
Be aware that non-pushdown conditions may adversely impact performance. Avoid using multiple conditions against the same resource as any non-pushdown condition will cause the entire OR statement to not be pushed down. If you need to insert permission conditions, be careful when adding an inline view, because adding them can cause performance problems if they are not compatible with your sources.

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

You can manage permission conditions on a per-role basis, but another approach is to add condition permissions to any authenticated role. By adding permissions in this way, the conditions are generalized for anyone using the `hasRole` , `user` , and other security functions. The advantage of this latter approach is that it provides you with a static row-based policy. As a result, your entire range of query plans can be shared among your users.

How you handle null values is up to you. You can implement ISNULL checks to ensure that null values are allowed when a column is *nullable*.

Limitations when using conditions

- Conditions on source tables that act as check constraints must currently not contain correlated subqueries.
- Conditions may not contain aggregate or windowed functions.
- Tables and procedures referenced via subqueries will still have row-based filters and column masking applied to them.

Note	Row-based filter conditions are enforced even for materialized view loads.

You should ensure that tables consumed to produce materialized views do not have row-based filter conditions on them that could affect the materialized view results.

Column masking
A permission against a fully qualified table/view/procedure column can also specify a mask and optionally a condition. When the query is submitted, the roles are consulted, and the relevant mask/condition information are combined to form a searched case expression to mask the values that would have been returned by the access. Unlike the CRUD allow actions defined above, the resulting masking effect is always applied — not just at the user query level. The condition and expression can be any valid SQL referencing the columns of the table/view/procedure. Procedure result set columns may be referenced as `proc.col` .

Application of column masks
Column masking is applied only against SELECTs. Column masking is applied logically after the affect of row-based security. However, because both views and source tables canb have row- and column-based security, the actual view-level masking can take place on top of source level masking. If the condition is specified along with the mask, then the effective mask expression affects only a subset of the rows: "CASE WHEN condition THEN mask ELSE column". Otherwise the condition is assumed to be TRUE, meaning that the mask applies to all rows.

If multiple roles specify a mask against a column, the mask order argument will determine their precedence from highest to lowest as part of a larger searched case expression. For example, a mask with the default order of 0 and a mask with an order of 1 would be combined as "CASE WHEN condition1 THEN mask1 WHEN condition0 THEN mask0 ELSE column".

Column masking considerations

Non-pushdown masking conditions/expressions can adversely impact performance, because their evaluation might inhibit pushdown of query constructs on top of the affected resource. In some circumstances the insertion of masking may require that the plan be altered with the addition of an inline view, which can result in poor performance if your sources are not compatible with the use of inline views.

In addition to managing masking on a per-role basis with the use of the order value, another approach is to specify masking in a single any authenticated role such that the conditions/expressions are generalized for all users/roles using the `hasRole` , `user` , and other such security functions. The advantage of the latter approach is that there is effectively a static masking policy in effect, such that all query plans can still be shared between users.

Column masking limitations

- If two masks have the same order value, it is not well defined what order they are applied in.
- Masks or their conditions cannot contain aggregate or windowed functions.
- Tables and procedures referenced via subqueries will still have row-based filters and column masking applied to them.

Note	Masking is enforced even for materialized view loads.

You should ensure that tables consumed to produce materialized views do not have masking on them that could affect the materialized view results.

Role mapping

Each Teiid data role can be mapped to any number of container roles or to 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. Note there is a difference permission type names between XML data roles and DDL grants - here SELECT, INSERT are referred to as READ and CREATE respectively.

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 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 virtual database.

By default, a system schema for ODBC metadata pg_catalog is also exposed. — however, that should be considered for general use.

Metadata visibility
The SYS system schema tables and procedures are always visible and accessible.

When data roles are in use, users can view only the tables, views, and procedure metadata entries that they have permissions to access. All columns of a key must be accessible for an entry to be visible.

Note	To make all metadata visible to any authenticated user, set the environment/system property `org.teiid.metadataRequiresPermission` to false.

Note	If you use data roles, visibility of entries can be affected by the caching of system metadata.

SYS schema

System schema for public information and actions.

SYS.Columns
This table supplies information about all the elements (columns, tags, attributes, etc) in the virtual database.

Column name	Type	Description
VDBName	string	VDB name
SchemaName	string	Schema name
TableName	string	Table name
Name	string	Element name (not qualified)
Position	integer	Position in group (1-based)
NameInSource	string	Name of element in source
DataType	string	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

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

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, and so forth) in the virtual database.

Column name	Type	Description
VDBName	string	VDB name
SchemaName	string	Schema Name
Name	string	Short group name
Type	string	Table type (Table, View, Document, …)
NameInSource	string	Name of this group in the source
IsPhysical	boolean	True if this is a source table
SupportsUpdates	boolean	True if group can be updated
UID	string	Group unique ID

Cardinality	integer	Approximate number of rows in the group
Description	string	Description
IsSystem	boolean	True if in system table
SchemaUID	string	Parent Schema unique ID

SYS.VirtualDatabases

This table supplies information about the currently connected virtual database, of which there is always exactly one (in the context of a connection).

Column name	Type	Description
Name	string	The name of the VDB
Version	string	The version of the VDB
Description	string	The description of the VDB
LoadingTimestamp	timestamp	The timestamp loading began
ActiveTimestamp	timestamp	The timestamp when the vdb became active.

SYS.spatial_sys_ref

See also the PostGIS Documentation

Column name	Type	Description
srid	integer	Spatial Reference Identifier
auth_name	string	Name of the standard or standards body
auth_srid	integer	SRID for the auth_name authority
srtext	string	Well-Known Text representation
proj4text	string	For use with the Proj4 library

SYS.GEOMETRY_COLUMNS

See also the PostGIS Documentation

Column name	Type	Description
F_TABLE_CATALOG	string	catalog name
F_TABLE_SCHEMA	string	schema name
F_TABLE_NAME	string	table name
F_GEOMETRY_COLUMN	string	column name

COORD_DIMENSION	integer	Number of coordinate dimensions
SRID	integer	Spatial Reference Identifier
TYPE	string	Geometry type name

Note: The `coord_dimension` and `srid properties` are determined from the `{http://www.teiid.org/translator/spatial/2015}coord_dimension` and `{http://www.teiid.org/translator/spatial/2015}srid` extension properties on the column. When possible, these values are set automatically by the relevant importer. If the values are not set, they will be reported as `2` and `0`, respectively. If client logic expects actual values, such as integration with GeoServer, you can set these values manually.

SYS.ArrayIterate
Returns a resultset with a single column with a row for each value in the array.

```
SYS.ArrayIterate(IN val object[]) RETURNS TABLE (col object)
```

Example: ArrayIterate

```
select array_get(cast(x.col as string[]), 2) from (exec arrayiterate((('a', 'b'),('c','d')))) x
```

This will produce two rows - 'b', and 'd'.

SYSADMIN schema

System schema for administrative information and actions.

SYSADMIN.Usage
The following table supplies information about how views and procedures are defined.

Column name	Type	Description
VDBName	string	VDB name
UID	string	Object UID
object_type	string	Type of object (StoredProcedure, ForeignProcedure, Table, View, Column, etc.)
Name	string	Object Name or parent name
ElementName	string	Name of column or parameter, may be null to indicate a table/procedure. Parameter level dependencies are currently not implemented.
Uses_UID	string	Used object UID
Uses_object_type	string	Used object type
Uses_SchemaName	string	Used object schema
Uses_Name	string	Used object name or parent name
Uses_ElementName	string	Used column or parameter name, may be null to indicate a table/procedure level dependency

Every column, parameter, table, or procedure referenced in a procedure or view definition will be shown as used. Likewise every column, parameter, table, or procedure referenced in the expression that defines a view column will be shown as used by that column. No dependency information is shown for procedure parameters. Column level dependencies are not yet inferred through intervening temporary or common tables.

Example: SYSADMIN.Usage

```
SELECT * FROM SYSADMIN.Usage
```

Recursive common table queries can be used to determine transitive relationships.

Example: Finding all incoming usage

```
with im_using as (
    select 0 as level, uid, Uses_UID, Uses_Name, Uses_Object_Type, Uses_ElementName
      from usage where uid = (select uid from sys.tables where name='table name' and schemaName='schema name')
    union all
    select level + 1, usage.uid, usage.Uses_UID, usage.Uses_Name, usage.Uses_Object_Type, usage.Uses_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

The following table supplies information about all the materailized views in the virtual database.

Column name	Type	Description
VDBName	string	VDB name
SchemaName	string	Schema Name
Name	string	Short group name
TargetSchemaName	string	Name of the materialized table schema. Will be null for internal materialization.
TargetName	string	Name of the materialized table
Valid	boolean	True if materialized table is currently valid. Will be null for external materialization.
LoadState	boolean	The load state, can be one of `NEEDS_LOADING` , `LOADING` , `LOADED` , `FAILED_LOAD` . Will be null for external materialization.
Updated	timestamp	The timestamp of the last full refresh. Will be null for external materialization.
Cardinality	integer	The number of rows in the materialized view table. Will be null for external materialization.

Valid, LoadState, Updated, and Cardinality may be checked for external materialized views with the `SYSADMIN.matViewStatus` procedure.

Example: SYSADMIN.MatViews

```
SELECT * FROM SYSADMIN.MatViews
```

SYSADMIN.VDBResources

The following table provides the current VDB contents.

Column Name	Type	Description
resourcePath	string	The path to the contents.

contents	blob	The contents as a blob.

Example: SYSADMIN.VDBResources

```
SELECT * FROM SYSADMIN.VDBResources
```

SYSADMIN.Triggers

The following table provides the triggers in the virtual database.

Column name	Type	Description
VDBName	string	VDB name
SchemaName	string	Schema Name
TableName	string	Table name
Name	string	Trigger name
TriggerType	string	Trigger Type
TriggerEvent	string	Triggering Event
Status	string	Is Enabled
Body	clob	Trigger Action (FOR EACH ROW …)
TableUID	string	Table Unique ID

Example: SYSADMIN.Triggers

```
SELECT * FROM SYSADMIN.Triggers
```

SYSADMIN.Views

The following table provides the views in the virtual database.

Column name	Type	Description
VDBName	string	VDB name
SchemaName	string	Schema Name
Name	string	View name
Body	clob	View Definition Body (SELECT …)
UID	string	Table Unique ID

Example: SYSADMIN.Views

```
SELECT * FROM SYSADMIN.Views
```

SYSADMIN.StoredProcedures

The following table provides the StoredProcedures in the virtual database.

Column name	Type	Description
VDBName	string	VDB name
SchemaName	string	Schema Name
Name	string	Procedure name
Body	clob	Procedure Definition Body (BEGIN …)
UID	string	Unique ID

Example: SYSADMIN.StoredProcedures

```
SELECT * FROM SYSADMIN.StoredProcedures
```

SYSADMIN.Requests

The following table provides active requests against the virtual database.

```
VDBName string(255) NOT NULL,
```

Column name	Type	Description
VDBName	string	VDB name
SessionId	string	session identifier
ExecutionId	long	execution identifier
Command	clob	The query being executed
StartTimestamp	timestamp	Start timestamp
TransactionId	string	transaction identifier as reported by the Transaction Manager
ProcessingState	string	processing state, can be one of PROCESSING, DONE, CANCELED
ThreadState	string	thread state, can be one of RUNNING, QUEUED, IDLE

SYSADMIN.Sessions

The following table provides the Sessions active for the virtual database.

Column name	Type	Description
VDBName	string	VDB name

SessionId	string	session identifier
UserName	string	username
CreatedTime	timestamp	timestamp of when the session was created
ApplicationName	string	application name as reported by the client
IPAddress	string	IP Address as reported by the client

SYSADMIN.Transactions

The following table provides the active Transactions.

Column name	Type	Description
TransactionId	string	transaction identifier as reported by the Transaction Manager
SessionId	string	session identifier if a session is currently associated with the transaction
StartTimestamp	timestamp	start time of the transaction
Scope	string	scope of the transaction, can be one of GLOBAL, LOCAL, REQUEST, INHERITED. INHERITED means that a Transaction was already associated with the calling thread (embedded usage).

Note: Transactions that are not associated with a given session will always be shown. Transactions that are associated with a session must be for a session with the current VDB.

SYSADMIN.isLoggable

Tests if logging is enabled at the given level and context.

```
SYSADMIN.isLoggable(OUT loggable boolean NOT NULL RESULT, IN level string NOT NULL DEFAULT 'DEBUG', IN context
string NOT NULL DEFAULT 'org.teiid.PROCESSOR')
```

Returns true if logging is enabled. level can be one of the log4j levels: OFF, FATAL, ERROR, WARN, INFO, DEBUG, TRACE. level defaults to 'DEBUG' and context defaults to 'org.teiid.PROCESSOR'

Example: isLoggable

```
IF ((CALL SYSADMIN.isLoggable(context=>'org.something'))
BEGIN
   DECLARE STRING msg;
   // logic to build the message ...
   CALL SYSADMIN.logMsg(msg=>msg, context=>'org.something')
END
```

SYSADMIN.logMsg

Log a message to the underlying logging system.

```
SYSADMIN.logMsg(OUT logged boolean NOT NULL RESULT, IN level string NOT NULL DEFAULT 'DEBUG', IN context string
 NOT NULL DEFAULT 'org.teiid.PROCESSOR', IN msg object)
```

Returns true if the message was logged. level can be one of the log4j levels: OFF, FATAL, ERROR, WARN, INFO, DEBUG, TRACE. The level defaults to 'DEBUG' and context defaults to 'org.teiid.PROCESSOR'. A null msg object will be logged as the string 'null'.

Example: logMsg

```
CALL SYSADMIN.logMsg(msg=>'some debug', context=>'org.something')
```

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

Table of Contents

SYSADMIN.refreshMatView

Full refresh/load of an internal materialized view. Returns integer RowsUpdated. -1 indicates a load is in progress, otherwise the cardinality of the table is returned. See the Caching Guide for more information.

See also SYSADMIN.loadMatView

```
SYSADMIN.refreshMatView(OUT RowsUpdated integer NOT NULL RESULT, IN ViewName string NOT NULL, IN Invalidate 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 information.

```
SYSADMIN.CREATE FOREIGN PROCEDURE refreshMatViewRow(OUT RowsUpdated integer NOT NULL RESULT, IN ViewName string
 NOT NULL, IN Key object NOT NULL, VARIADIC KeyOther object)
```

Example: SYSADMIN.refreshMatViewRow

The materialized view `SAMPLEMATVIEW` has 3 rows under the `TestMat` Model as below:

```
+------+-----+-----+-----+
| id   | a   | b   | c   |
+------+-----+-----+-----+
| 100  | a0  | b0  | c0  |
| 101  | a1  | b1  | c1  |
| 102  | a2  | b2  | c2  |
+------+-----+-----+-----+
```

Assuming the primary key only contains one column, id, update the second row:

```
EXEC SYSADMIN.refreshMatViewRow('TestMat.SAMPLEMATVIEW', '101')
```

Assuming the primary key contains more columns, a and b, update the second row:

```
EXEC SYSADMIN.refreshMatViewRow('TestMat.SAMPLEMATVIEW', '101', 'a1', 'b1')
```

SYSADMIN.refreshMatViewRows

Refreshes rows in an internal materialized view.

Returns integer RowsUpdated. -1 indicates the materialized table is currently invalid. Any row that does not exist in the live data query or in the materialized table will not count toward the RowsUpdated. For more information, see the Teiid Caching Guide.

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

Example: SYSADMIN.refreshMatViewRows

Continuing use the `SAMPLEMATVIEW` in Example of SYSADMIN.refreshMatViewRow. Assuming the primary key only contains one column, id, update all rows:

```
EXEC SYSADMIN.refreshMatViewRows('TestMat.SAMPLEMATVIEW', ('100',), ('101',), ('102',))
```

Assuming the primary key comtain more columns, id, a and b compose of the primary key, update all rows:

```
EXEC SYSADMIN.refreshMatViewRows('TestMat.SAMPLEMATVIEW', ('100', 'a0', 'b0'), ('101', 'a1', 'b1'), ('102', 'a2
', 'b2'))
```

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'
)
```

The preceding example will set the property 'some name'='some value' on table tab.

Note	The use of this procedure will not trigger replanning of associated prepared plans.

Properties from built-in teiid_* namespaces can be set using the the short form - namespace:key form.

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 information.

SYSADMIN.matViewStatus
matViewStatus is used to retrieve the status of materialized views via schemaName and viewName.

Returns tables which contains TargetSchemaName, TargetName, Valid, LoadState, Updated, Cardinality, LoadNumber, OnErrorAction.

```
SYSADMIN.matViewStatus(IN schemaName string NOT NULL, IN viewName string NOT NULL) RETURNS TABLE (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 information.

```
SYSADMIN.loadMatView(IN schemaName string NOT NULL, IN viewName string NOT NULL, IN invalidate boolean NOT NULL
 DEFAULT 'false') RETURNS integer
```

Example: loadMatView

```
exec SYSADMIN.loadMatView(schemaName=>'TestMat',viewname=>'SAMPLEMATVIEW', invalidate=>'true')
```

SYSADMIN.updateMatView

The updateMatView procedure is used to update a subset of an internal or external materialized table based on the refresh criteria.

The refresh criteria might reference the view columns by qualified name, but all instances of `.` in the view name will be replaced by `_`, because an alias is actually being used.

Returns integer RowsUpdated. -1 indicates the materialized table is currently invalid. And -3 indicates there was an exception when performing the update. See the Caching Guide for more information.

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

SYSADMIN.cancelRequest

Cancel the user request identified by execution id for the given session.

See also SYSADMIN.REQUESTS

```
SYSADMIN.cancelRequest(OUT cancelled boolean NOT NULL RESULT, IN SessionId string NOT NULL, IN executionId long
 NOT NULL)
```

Example: Cancel

```
CALL SYSADMIN.cancelRequest('session id', 1)
```

SYSADMIN.terminateSession

Terminate the session with the given identifier.

See also SYSADMIN.SESSIONS

```
SYSADMIN.terminateSession(OUT terminated boolean NOT NULL RESULT, IN SessionId string NOT NULL)
```

Example: Termination

```
CALL SYSADMIN.terminateSession('session id')
```

SYSADMIN.terminateTransaction

Terminate the transaction associated with a session by marking the transaction as rollback only.

See also SYSADMIN.TRANSACTIONS

```
SYSADMIN.terminateTransaction(IN sessionid string NOT NULL)
```

Note	You cannot only cancel transactions that are associated with a session.

Example: Terminate

```
CALL SYSADMIN.terminateTransaction('session id')
```

Translators

Teiid uses the Teiid Connector Architecture (TCA), which provides 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 available for pushdown and the ability to import metadata from the external system.

A Translator is the heart of the TCA and acts as the bridge logic between Teiid and an external system.

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, such as the Derby translator, base execution properties are already tuned to match the source. In most cases the user will not need to adjust their values.

Table 1. **Base execution properties - shared by all translators**

Name	Description	Default
Immutable	Set to `true` to indicate that the source never changes. The transactional capability is reported as NONE, and update commands will fail.	false
RequiresCriteria	Set to `true` to indicate that source SELECT/UPDATE/DELETE queries require a where clause.	false
SupportsOrderBy	Set to `true` to indicate that the ORDER BY clause can be used.	false
SupportsOuterJoins	Set to `true` to indicate that OUTER JOINs can be used.	false
SupportsFullOuterJoins	If `SupportsOuterJoins` is set to `true` , `true` indicates that FULL OUTER JOINs can be used.	false
SupportsInnerJoins	Set to `true` to indicate that INNER JOINs can be used.	false
SupportedJoinCriteria	If join capabilities are enabled, defines the criteria that can be used as the join criteria. May be one of (ANY, THETA, EQUI, or KEY).	ANY

MaxInCriteriaSize	If the use of `IN` criteria is enabled, specifies the maximum number of `IN` entries per predicate. `-1` indicates no limit.	-1
MaxDependentInPredicates	If the use of `IN` criteria is enabled, defines what the maximum number of predicates that can be used for a dependent join. Values less than 1 indicate to use only one `IN` predicate per dependent value pushed (which matches the pre-7.4 behavior).	-1
DirectQueryProcedureName	If `SupportsDirectQueryProcedure` is set to `true` for the translator, this property indicates the name of the procedure.	native
SupportsDirectQueryProcedure	Set to `true` to indicate that direct execution of commands is available for the translator.	false
ThreadBound	Set to `true` to indicate the translator's Executions should be processed by only a single thread	false
CopyLobs	If `true`, then returned large object (LOB) data (clob, blob, sql/xml) is copied by the engine in a memory safe manner. Use this option if the source does not provide memory safe LOBS or if you want to disconnect LOBS from the source connection.	false
TransactionSupport	The highest level of transaction capability. Used by the engine as a hint to determine if a transaction is needed for `autoCommitTxn=DETECT` mode. Can be one of XA, NONE, or LOCAL. If XA or LOCAL then access under a transaction will be serialized.	XA

Note	Only a subset of the available metadata can be set through execution properties on the base ExecutionFactory. All methods are available on the BaseDelegatingExecutionFactory.

There are no base importer settings.

Override execution properties
For all translators, you can override Execution Properties in the main vdb file.

Example: Overriding a translator property

.

```
CREATE FOREIGN DATA WRAPPER "oracle-override" TYPE oracle OPTIONS (RequiresCriteria 'true');

CREATE SERVER ora FOREIGN DATA WRAPPER "oracle-override" OPTIONS ("resource-name" 'java:/oracle');

CREATE SCHEMA ora SERVER ora;

SET SCHEMA ora;
```

```
IMPORT FROM SERVER ora INTO ora;
```

Or as an XML vdb:

```
<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 preceding example overrides the *oracle* translator and sets the *RequiresCriteria* property to true. The modified translator is only available in the scope of this VDB. As many properties as desired may be overriden together.

See also VDB Definition.

Parameterizable native queries

In some situations the `teiid_rel:native-query` `property and native procedures accept parameterizable strings that can positionally reference IN parameters. A parameter reference has the form `$integer`, for example, `$1`. Note that one-based indexing is used and that only IN parameters may be referenced. Dollar-sign integer is therefore reserved, but may be escaped with another `$``, for example, `$$1`. The value will be bound as a prepared value or a literal is a source specific manner. The native query must return a result set that matches the expectation of the calling procedure.

For example the native-query `select c from g where c1 = $1 and c2 = '$$1'` results in a JDBC source query of `select c from g where c1 = ? and c2 = '$1'`, where `?`` will be replaced with the actual value bound to parameter 1.

General import properties

Several import properties are shared by all translators.

When specifying an importer property, it must be prefixed with `importer.`. For example, `importer.tableTypes`.

Name	Description	Default
autoCorrectColumnNames	Replace any usage of `.` in a column name with `_` as the period character is not valid in Teiid 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_`.	

S3 translator

The Simple Storage Service (S3) translator, known by the type name *amazon-s3*, exposes stored procedures to leverage Amazon S3 object resources. The translator works with a range of S3-compatible data sources, including Ceph Storage, Google Cloud Storage *buckets*, MinIO, and NooBaa.

This translator is typically used with the `TEXTTABLE` or `XMLTABLE` functions to consume CSV or XML formatted data, or to read Microsoft Excel files or other object files that are stored in S3. The S3 translator can access Amazon S3 by using an AWS access key ID and secret access key.

Usage

In the following example, a virtual database reads a CSV file with the name `g2.txt` from an Amazon S3 bucket called `teiidbucket`:

```
e1,e2,e3
5,'five',5.0
6,'six',6.0
7,'seven',7.0
```

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<vdb name="example" version="1">
    <model name="s3">
        <source name="web-connector" translator-name="user-s3" connection-jndi-
name="java:/amazon-s3"/>
    </model>
    <model name="Stocks" type="VIRTUAL">
        <metadata type="DDL"><![CDATA[
        CREATE VIEW G2 (e1 integer, e2 string, e3 double,PRIMARY KEY (e1))
            AS  SELECT SP.e1, SP.e2,SP.e3
                FROM (EXEC s3.getTextFile(name=>'g2.txt')) AS f,
                TEXTTABLE(f.file COLUMNS e1 integer, e2 string, e3 double HEADER)
AS SP;
        ]]> </metadata>
    </model>
    <translator name="user-s3" type="amazon-s3">
      <property name="accesskey" value="xxxx"/>
      <property name="secretkey" value="xxxx"/>
      <property name="region" value="us-east-1"/>
      <property name="bucket" value="teiidbucket"/>
    </translator>
</vdb>
```

Execution properties

Use the translator override mechanism to supply the following properties.

Name	Description	Default
Encoding	The encoding that should be used for CLOBs returned by the getTextFiles procedure. The value should match an encoding known to the JRE.	The system default encoding.

Accesskey	Amazon security access key. Log in to Amazon console to find your security access key. When provided, this becomes the default access key.	n/a
Secretkey	Amazon security secret key. Log in to Amazon console to find your security secret key. When provided, this becomes the default secret key.	n/a
Region	Amazon region to be used with the request. When provided, this will be default region used.	US-EAST-1
Bucket	Amazon S3 bucket name. If provided, this will serve as default bucket to be used for all the requests	n/a
Encryption	When server-side encryption with customer-provided encryption keys (SSE-C) is used, the key is used to define the "type" of encryption algorithm used. You can configure the translator to use the AES-256 or AWS-KMS encryption algorithms. If provided, this will be used as default algorithm for all "get" based calls.	n/a
Encryptionkey	When SSE-C type encryption used, where customer supplies the encryption key, this key will be used for defining the "encryption key". If provided, this will be used as default key for all "get" based calls.	n/a

Tip	For information about setting properties, see *Override execution property* in Translators, and review the examples in the sections that follow.

If you are using an S3 service other than AWS, then you need to set the EndPoint property on the associated source to the service URL, i.e. http://my-minio:9000

Procedures exposed by translator
When you add the a model (schema) like above in the example, the following procedure calls are available for user to execute against Amazon S3.

Note	`bucket` , `region` , `accesskey` , `secretkey` , `encryption` and `encryptionkey` are optional or nullable parameters in most of the methods provided. Provide them only if they are not already configured by using translator override properties as shown in preceding example.

getTextFile(…)
Retrieves the given named object as a text file from the specified bucket and region by using the provided security credentials as CLOB.

```
getTextFile(string name NOT NULL, string bucket, string region,
   string endpoint, string accesskey, string secretkey,string encryption, string encryptionkey, boolean stream
default false)
   returns TABLE(file blob, endpoint string, lastModified string, etag string, size long);
```

	`endpoint` is optional. When provided the endpoint URL is used instead of the one constructed by the supplied

Note	properties. Use `encryption` and `encryptionkey` only in when server side security with customer supplied keys (SSE-C) in force.

If the value of `stream` is true, then returned LOBs are read only once and are not typically buffered to disk.

Examples

```
exec getTextFile(name=>'myfile.txt');

SELECT SP.e1, SP.e2,SP.e3, f.lastmodified
   FROM (EXEC getTextFile(name=>'myfile.txt')) AS f,
   TEXTTABLE(f.file COLUMNS e1 integer, e2 string, e3 double HEADER) AS SP;
```

getFile(…)

Retrieves the given named object as binary file from specified bucket and region using the provided security credentials as BLOB.

```
getFile(string name NOT NULL, string bucket, string region,
   string endpoint, string accesskey, string secretkey, string encryption, string encryptionkey, boolean stream
 default false)
   returns TABLE(file blob, endpoint string, lastModified string, etag string, size long)
```

Note	`endpoint` is optional. When provided the endpoint URL is used instead of the one constructed by the supplied properties. Use `encryption` and `encryptionkey` only in when server side security with customer supplied keys (SSE-C) in force.

If the value of `stream` is true, then returned lOBs are read once and are not typically buffered to disk.

Examples

```
exec getFile(name=>'myfile.xml', bucket=>'mybucket', region=>'us-east-1', accesskey=>'xxxx', secretkey=>'xxxx')
;

select b.* from (exec getFile(name=>'myfile.xml', bucket=>'mybucket', region=>'us-east-1', accesskey=>'xxxx', 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 following procedure signature, the *contents* parameter can be any of the LOB types.

```
call saveFile(string name NOT NULL, string bucket, string region, string endpoint,
   string accesskey, string secretkey, contents object)
```

Note	You cannot use `saveFile` to stream or chunk uploads of a file's contents. If you try to load very large objects, out-of-memory issues can result. You cannot configure `saveFile` to use SSE-C encryption.

Examples

```
exec saveFile(name=>'g4.txt', contents=>'e1,e2,e3\n1,one,1.0\n2,two,2.0');
```

deleteFile(…)

Delete the named object from the bucket.

```
call deleteFile(string name NOT NULL, string bucket, string region, string endpoint, string accesskey, string s
ecretkey)
```

Examples

```
exec deleteFile(name=>'myfile.txt');
```

list(…)
Lists the contents of the bucket using a v2 list request.

```
call list(string bucket, string region, string accesskey, string secretkey, nexttoken string)
    returns Table(result clob)
```

listv1(…)
Lists the contents of the bucket using a v1 list request.

```
call listv1(string bucket, string region, string accesskey, string secretkey)
    returns Table(result clob)
```

The result for either is an XML file which resembles:

```
<?xml version="1.0" encoding="UTF-8"?>/n
<ListBucketResult
    xmlns="http://s3.amazonaws.com/doc/2006-03-01/">
    <Name>teiidbucket</Name>
    <Prefix></Prefix>
    <KeyCount>1</KeyCount>
    <MaxKeys>1000</MaxKeys>
    <IsTruncated>false</IsTruncated>
    <Contents>
        <Key>g2.txt</Key>
        <LastModified>2017-08-08T16:53:19.000Z</LastModified>
        <ETag>&quot;fa44a7893b1735905bfcce59d9d9ae2e&quot;</ETag>
        <Size>48</Size>
        <StorageClass>STANDARD</StorageClass>
    </Contents>
</ListBucketResult>
```

You can parse this into a view by using a query similar to the one in the following example:

```
select b.* from (exec list(bucket=>'mybucket', region=>'us-east-1')) as a,
 XMLTABLE(XMLNAMESPACES(DEFAULT 'http://s3.amazonaws.com/doc/2006-03-01/'), '/ListBucketResult/Contents'
 PASSING XMLPARSE(CONTENT a.result WELLFORMED) COLUMNS Key string, LastModified string, ETag string, Size string
,
 StorageClass string,   NextContinuationToken string PATH '../NextContinuationToken') as b;
```

If all properties (`bucket` , `region` , `accesskey` , and `secretkey`) are defined as translator override properties, you can run the following simple query:

```
SELECT * FROM Bucket
```

Note	If there are more then 1000 object in the bucket, then the value 'IsTruncated' will be true. v2 support for a Bucket list with continuation support can be automated in Teiid with an enhancement request.

JCA Resource Adapter

The resource adapter for this translator provided through "Web Service Data Source", Refer to the Teiid Administrator’s Guide for configuration information.

Amazon SimpleDB Translator

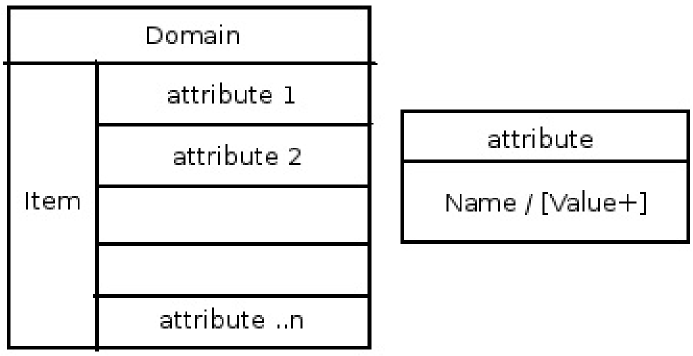
The Amazon SimpleDB Translator, known by the type name *simpledb*, exposes querying functionality to Amazon SimpleDB Data Sources.

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
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 provides a restrictive set of capabilities for SELECT statements, including: comparison predicates, IN predicates, LIMIT and ORDER BY. The translator also works with Insert, update, and delete statements.

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'))
```

Note	Direct Queries 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. You can use direct query with `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="5">Key</th><th rowspan="3">Value</th></tr>
<tr><td rowspan="2">Row ID</td><td colspan="3">Column</td><td rowspan="2">Timestamp</td></tr>
<tr><td>Family</td><td>Qualifier</td><td>Visibility</td></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>
```

JCA Resource Adapter

The Teiid specific Solr Resource Adapter should be used with this translator. See Solr Data Sources or see an example in "<jboss-as>/docs/teiid/datasources/solr" 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

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

```
Datastores
    └── Namespaces
            └── Keyspaces
                   └── Documents
```

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, ar 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')
```

Delegator translators

You can use the *delegator* translator, which is available in the core Teiid installation, to modify the capabilities of a existing translator. Often times for debugging purposes, or in special situations, you might want to turn certain capabilities of a translator on or off. For example, say that the latest version of a Hive database supports the `ORDER BY` construct, but the current Teiid version of the Hive translator does not. You could use the delegator translator to enable `ORDER BY` compatibility without actually writing any code. Similarly, you could do the reverse, and turn off certain capabilities to produce a better plan.

To use the delegator translator, you must define it in the DDL. The following example shows how to override the "hive" translator and turn off the `ORDER BY` capability.

```
CREATE DATABASE myvdb;
USE DATABASE myvdb;
CREATE FOREIGN DATA WRAPPER "hive-delegator" TYPE delegator OPTIONS (delegateName 'hive', supportsOrderBy 'fals
e');
CREATE SERVER source FOREIGN DATA WRAPPER "hive-delegator" OPTIONS ("resource-name" 'java:hive-ds');
CREATE SCHEMA mymodel SERVER source;
SET SCHEMA mymodel;
IMPORT FROM SERVER source INTO mymodel;
```

```
<?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>
```

For more information about the translator capabilities that you can override by using execution properties, see *Translator_Capabilities* in the Translator Development Guide. The preceding example shows how you might modify the default `ORDER BY` compatibility of the Hive translator.

Extending the delegator translator

You can create a delegating translator by extending the `org.teiid.translator.BaseDelegatingExecutionFactory`. After your classes are packaged as a custom translator, you can wire another translator instance into your delegating translator at runtime in order to intercept all of the calls to the delegate. This base class does not provide any functionality on its own, other than delegation. You can hard code capabilities into the translator instead of defining them as part of the DDL configuration. You can also override methods to provide alternate behavior.

Table 1. **Execution properties**

Name	Description	Default
delegateName	Translator instance name to delegate to.	n/a
cachePattern	Regex pattern of queries that should be cached using the translator caching API.	n/a
cacheTtl	Time to live in milliseconds for queries matching the cache pattern.	n/a

For example, if you use the *oracle* translator in your virtual database, and you want to intercept calls that go through the translator, you could write a custom delegating translator, as in the following example:

```
@Translator(name="interceptor", description="interceptor")
public class InterceptorExecutionFactory extends org.teiid.translator.BaseDelegatingExecutionFactory{
    @Override
    public void getMetadata(MetadataFactory metadataFactory, C conn) throws TranslatorException {
        // do intercepting code here..

        // If you want call the original delegate, do not call if do not need to.
        // but if you did not call the delegate fullfill the method contract
        super.getMetadata(metadataFactory, conn);

        // do more intercepting code here..
    }
}
```

You could then deploy this translator in the Teiid engine. Then in your DDL file, define an interceptor translator as in the following example:

```
CREATE DATABASE myvdb VERSION '1';
USE DATABASE myvdb VERSION '1';
CREATE FOREIGN DATA WRAPPER "oracle-interceptor" TYPE interceptor OPTIONS (delegateName 'oracle');
CREATE SERVER source FOREIGN DATA WRAPPER "oracle-interceptor" OPTIONS ("resource-name" 'java:oracle-ds');
CREATE SCHEMA mymodel SERVER source;
SET SCHEMA mymodel;
IMPORT FROM SERVER source INTO mymodel;
```

```
<?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 the VDB. Future calls going into this VDB model's translator are 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 resources. The translator is typically used with the `TEXTTABLE` or `XMLTABLE` functions to consume CSV or XML formatted data.

Table 1. **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. For more information, see *TO_CHARS* and *TO_BYTES* in String functions.	The system default encoding.
ExceptionIfFileNotFound	Throw an exception in getFiles or getTextFiles if the specified file/directory does not exist.	true

Tip	For information about how to set properties, see the following example, and *Override execution properties* in Translators.

Example: Virtual datbase DDL override

```
CREATE SERVER "file-override"
    FOREIGN DATA WRAPPER file
    OPTIONS(
        Encoding 'ISO-8859-1', "ExceptionIfFileNotFound" false
    );

CREATE SCHEMA file SERVER "file-override";
```

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>
```

getFiles

```
getFiles(String pathAndPattern) returns
TABLE(file blob, filePath string, lastModified timestamp, created timestamp, size long)
```

Retrieve all files as BLOBs matching the given path and pattern.

```
call getFiles('path/*.ext')
```

If the path is a directory, then all files in the directory are returned. If the path matches a single file, the file is returned.

The `*` character is treated as a wildcard to match any number of characters in the path name. Zero or matching files will be returned.

If `*' is not used, and if the path doesn't exist and `ExceptionIfFileNotFound` is true, then an exception is raised.

getTextFiles

```
getTextFiles(String pathAndPattern) returns
TABLE(file clob, filePath string, lastModified timestamp, created timestamp, size long)
```

Note	The size reports the number of bytes.

Retrieve all files as CLOBs matching the given path and pattern.

```
call getTextFiles('path/*.ext')
```

Retrieves the same files `getFiles` , but with the difference that the results are CLOB values that use the encoding execution property as the character set.

saveFile
Save the CLOB, BLOB, or XML value to given path

```
call saveFile('path', value)
```

deleteFile
Delete the file at the given path

```
call deleteFile('path')
```

The path should reference an existing file. If the file does not exist and `ExceptionIfFileNotFound` is true, then an exception will be thrown. An exception is also thrown if the file cannot be deleted.

JCA resource adapter
For configuration information, see File Data Source, the FTP Data Source, and the Administrator's Guide in general.

Note	Native queries This feature is not applicable for the File translator.

Note	Direct query procedure This feature is not applicable for the File translator.

Google spreadsheet translator

The *google-spreadsheet* translator is used to connect to a Google Sheets spreadsheet.

The query approach expects that the data in the worksheet has the following characteristics:

- All columns that contains data can be queried.
- Any column with an empty cell has the value retrieved as null. However, differentiating between null string and empty string values may not always be possible as Google treats them interchangeably. Where possible, the translator may provide a warning or throw an exception if it cannot differentiate between null and empty strings.
- If the first row is present and contains string values, then the row is assumed to represent the column labels.

If you are using the default native metadata import, the metadata for your Google account (worksheets and information about columns in worksheets) is loaded upon translator start up. If you make any changes in data types, it is advisable to restart your virtual database.

The translator can submit queries against a single sheet only. It provides ordering, aggregation, basic predicates, and most of the functions available in the spreadsheet query language.

The google-spreadsheet translator does not provide importer settings, but it can provide metadata for VDBs.

Warning	If you remove all data rows from a sheet with a header that is defined in Teiid, you can no longer access the sheet through Teiid. The Google API will treat the header as a data row at that point, and queries to it will no longer be valid.

Warning	Non-string fields are updated using the canonical Teiid SQL value. In cases where the spreadsheet is using a non-conforming locale, consider disallowing updates. For more information, see TEIID-4854 and the following information about the `allTypesUpdatable` import property.

Importer properties

- *allTypesUpdatable*- Set to true to mark all columns as updatable. Set to false to enable update only on string or Boolean columns that are not affected by TEIID-4854. Defaults to true.

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. For more information, see *Parameterizable native queries* in Translators. The procedure will invoke the native-query similar to an native procedure call, with the benefits that the query is predetermined, and that result column types are known, rather than requiring the use of ARRAYTABLE or similar functionality. For more information, see the *Select* section that follows.

Note	Direct query procedure This feature is turned off by default, because of the security risk in permitting any command to execute against the data source. To enable this feature, set the property *SupportsDirectQueryProcedure* to true. For more information, see *Override execution properties* in Translators.

Tip	By default the name of the procedure that executes the queries directly is called **native**. You can change its name by overriding the execution property *DirectQueryProcedureName*. For more information, see *Override execution properties* in Translators.

The Google spreadsheet translator provides a procedure to execute any ad-hoc query directly against the source without any Teiid parsing or resolving. Because the metadata of this procedure's execution results are not known to Teiid, they are returned as an object array. You can use ARRAYTABLE to construct tabular output for consumption by client applications. For more information, see ARRAYTABLE.

Teiid exposes this procedure with a simple query structure as shown in the following example:

Select example

```
SELECT x.* FROM (call google_source.native('worksheet=People;query=SELECT A, B, C')) w,
 ARRAYTABLE(w.tuple COLUMNS "id" string , "type" string, "name" String) AS x
```

The first argument takes semicolon-separated (;) name-value pairs of the following properties to execute the procedure:

Property	Description	Required
worksheet	Google spreadsheet name.	yes
query	Spreadsheet query.	yes
limit	Number of rows to fetch.	no
offset	Offset of rows to fetch from limit or beginning.	no

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.
- 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)

JCA Resource Adapter

The resource adapter for this translator is a Infinispan Data Source.

Native Queries

Note	This feature is not applicable for the Infinispan translator.

Direct Query Procedure

Note	This feature is not applicable for the Infinispan translator.

JDBC translators

The JDBC translators bridge the SQL semantics 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 relational databases.

Usage
Usage of a JDBC source is straight-forward. Using Teiid SQL, the source can 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, you can define a custom translator that does what you need. For more information, see Translator Development.

Type conventions
UID types including UUID, GUID, or UNIQUEIDENTIFIER are typically mapped to the Teiid string type. JDBC data sources treat UID strings as non-case sensitive, but they are case-sensitive in Teiid. If the source does not support the implicit conversion to the string type, then usage in functions that expect a string value might fail at the source.

The following table lists the execution properties that are shared by all JDBC translators.

Table 1. Execution properties — Shared by all JDBC translators

Name	Description	Default
DatabaseTimeZone	The time zone of the database. Used when fetching date, time, or timestamp values.	The system default time zone
DatabaseVersion	The specific database version. Used to further tune the use of pushdown operations.	The base compatible version, or the version that is derived from the DatabaseMetadata.getDatabaseProductVersion string. Automatic detection requires a connection. If there are circumstances where you are getting an exception due to capabilities being unavailable (for example, because a connection is not available), then set `DatabaseVersion` property. Use the `JDBCExecutionFactory.usesDatabaseVersion()`` method to control whether your translator requires a connection to determine capabilities.
TrimStrings	`true` trims 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` indicates that PreparedStatements should be used and that literal values in the source query should be replace with bind variables. If `false` only LOB values will trigger the use of PreparedStatements.	true
UseCommentsInSourceQuery	This will embed a leading comment with session/request id in the source SQL for informational purposes. Can be customized with the CommentFormat property.	false
CommentFormat	MessageFormat string to be used if `UseCommentsInSourceQuery` is enabled. You can set the format to one of the following values: • 0 - Session ID string. • 1 - Parent request ID string. • 2 - Request part ID string. • 3 - Execution count ID string. • 4 - User name string. • 5 - VDB name string. • 6 - VDB version integer. • 7 - Is transactional boolean.	`/*teiid sessionid:\{0}, requestid:\{1}.\{2}*/`
MaxPreparedInsertBatchSize	The max size of a prepared insert batch.	2048
StructRetrieval	Specify one of the following Struct retrieval modes: • OBJECT - `getObject` value returned. • COPY - Returned as a `SerialStruct` . • ARRAY - Returned as an array.	OBJECT
EnableDependentJoins	Allow dependent join pushdown for sources that use temporary tables (DB2, Derby, H2, HSQL 2.0+, MySQL 5.0+, Oracle, PostgreSQL, SQLServer, SQP IQ, Sybase).	false

Importer properties — Shared by all JDBC translators

When specifying the importer property, it must be prefixed with `importer.` . Example: `importer.tableTypes`

Name	Description	Default
catalog	See DatabaseMetaData.getTables [1]	null
schemaName	Recommended setting to import from a single schema. The schema name will be converted into an escaped pattern,overriding schemaPattern if it is also set.	null
schemaPattern	See DatabaseMetaData.getTables [1]	null
tableNamePattern	See DatabaseMetaData.getTables [1]	null
procedureNamePattern	See DatabaseMetaData.getProcedures [1]	null
tableTypes	Comma separated list — without spaces — of imported table types. See `DatabaseMetaData.getTables` [1]	null
excludeTables	A case-insensitive regular expression that when matched against a fully qualified table name [2] will exclude it from import. Applied after table names are retrieved. Use a negative look-ahead (?!<inclusion pattern>).* to act as an inclusion filter.	null
excludeProcedures	A case-insensitive regular expression that when matched against a fully qualified procedure name [2] will exclude it from import. Applied after procedure names are retrieved. Use a negative look-ahead (?!<inclusion pattern>).* to act as an inclusion filter.	null
importKeys	`true` to import primary and foreign keys. NOTE: Foreign keys to tables that are not imported will be ignored.	true
autoCreateUniqueConstraints	`true` to create a unique constraint if one is not found for a foreign keys	true
importIndexes	`true` to import index/unique key/cardinality information	false
importApproximateIndexes	`true` to import approximate index information. See DatabaseMetaData.getIndexInfo [1]. WARNING: Setting to `false` may cause lengthy import times.	true

importProcedures	`true` to import procedures and procedure columns - Note that it is not always possible to import procedure result set columns due to database limitations. It is also not currently possible to import overloaded procedures.	false
importSequences	`true` to import sequences. Compatible only with Db2, Oracle, PostgreSQL, SQL Server, and H2. A matching sequence will be imported to a 0-argument 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 only the object name as the Teiid name. It is expected that all objects will come from the same foreign schema. When `true` (not recommended) the Teiid name will be formed using the catalog and schema names as directed by the `useCatalogName` and `useQualifiedName` properties, and it will be allowed for objects to come from multiple foreign schema. This option does not affect the name in source property.	false (only change when importing from multiple foreign schema).
useQualifiedName	`true` will use name qualification for both the 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 you set this option to `false` , it can lead to objects with duplicate names when importing from multiple schemas, which results in an exception.	true (rarely needs changed)
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 such as HSQL that do not use the catalog concept, but return a non-null/non-empty catalog name in their metadata.	true (rarely needs changed)

widenUnsignedTypes	`true` to convert unsigned types to the next widest type. For example, SQL Server reports `tinyint` as an unsigned type. With this option enabled, `tinyint` would be imported as a short instead of a byte.	true
useIntegralTypes	`true` to use integral types rather than decimal when the scale is 0.	false
quoteNameInSource	`false` will override the default and direct 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 return a statistical index.	false
importStatistics	`true` will use database dependent logic to determine the cardinality if none is determined. Not available for all database types — currently available for Oracle and MySQL only.	false
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 import only tables and views from my-schema.

```
SET SCHEMA ora;

IMPORT FOREIGN SCHEMA "my-schema" FROM SERVER ora INTO ora OPTIONS ("importer.tableTypes" 'TABLE,VIEW');
```

Or in an xml vdb:

```
<model ...
```

```
  <property name="importer.tableTypes" value="TABLE,VIEW"/>
  <property name="importer.schemaName" value="my-schema"/>
  ...
</model>
```

For more information about importer settings, see Virtual databases.

Native queries
Physical tables, functions, and procedures may optionally have native queries associated with them. No validation of the native query is performed, it is simply used in a straight-forward manner to generate the source SQL. For a physical table setting the `teiid_rel:native-query` extension metadata will execute the native query as an inline view in the source query. This feature should only be used against sources that provide inline views. The native query is used as is and is not treated as a parameterized string. For example, on a physical table `y` with `` nameInSource="x"` `` and `teiid_rel:native-query="select c from g"` , the 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 . For more information, see *Parameterizable native queries* in Translators. The `teiid_rel:non-prepared` extension metadata property can be set to `false` to turn off parameter binding.

Be careful in setting this option, because inbound allows for SQL injection attacks if not properly validated. The native query does not need to call a stored procedure. Any SQL that returns a result set that positionally matches the result set that is expected by the physical stored procedure metadata will work. For example on a stored procedure `x` with ``teiid_rel:native-query="select c from g where c1 = $1 and c2 = `$$1"'`` , the 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 inherent security risk in allowing any command to be run against the source. To enable this feature, override the execution property called *SupportsDirectQueryProcedure* and set it to `true` . For more information, see *Override execution properties* in Translators.

By default, the name of the procedure that executes the queries directly is **native**. To change the name, override the execution property *DirectQueryProcedureName*.

The JDBC translator provides a procedure to execute any ad-hoc SQL query directly against the source without 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. For more information, see arraytable.

SELECT example

```
SELECT x.* FROM (call jdbc_source.native('select * from g1')) w,
 ARRAYTABLE(w.tuple COLUMNS "e1" integer , "e2" string) AS x
```

INSERT example

```
SELECT x.* FROM (call jdbc_source.native('insert into g1 (e1,e2) values (?, ?)', 112, 'foo')) w,
 ARRAYTABLE(w.tuple COLUMNS "update_count" integer) AS x
```

UPDATE example

```
SELECT x.* FROM (call jdbc_source.native('update g1 set e2=? where e1 = ?','blah', 112)) w,
 ARRAYTABLE(w.tuple COLUMNS "update_count" integer) AS x
```

DELETE example

```
SELECT x.* FROM (call jdbc_source.native('delete from g1 where e1 = ?', 112)) w,
 ARRAYTABLE(w.tuple COLUMNS "update_count" integer) AS x
```

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 Translators information.

The Actian Vector translator, known by the type name ***actian-vector***, is for use with Actian Vector in Hadoop.

Download the JDBC driver at http://esd.actian.com/platform. Note the port number in the connection URL is "AH7", which maps to 16967.

Apache Phoenix Translator (phoenix)

Also see common JDBC Translators information.

The Apache Phoenix translator, known by the type name ***phoenix***, exposes querying functionality to HBase tables. Apache Phoenix is a JDBC SQL interface for HBase that is required for this translator as it pushes down commands into Phoenix SQL.

The translator is also known by the deprecated name ***hbase***. The name change reflects that the translator is specific to Phoenix and that there could be other translators introduced in the future to connect to HBase.

Do not use the `DatabaseTimezone` property with this translator.

The HBase translator cannot process Join commands. Phoenix uses the HBase Table Row ID as the Primary Key. This Translator is developed with Phoenix 4.3 or greater for HBase 0.98.1 or greater.

Note	The translator implements `INSERT` / `UPDATE` through the Phoenix `UPSERT` operation. This means you can see different behavior than with standard `INSERT` / `UPDATE`. For example, repeated inserts will not throw a duplicate key exception, but will instead update the row in question.

Note	Due to Phoenix driver limitations, the importer does not look for unique constraints, and defaults to not importing foreign keys.

Note	The translator can process SQL `OFFSET` arguments and other features starting with Phoenix 4.8. The Phoenix driver hard codes the server version in `PhoenixDatabaseMetaData`, and does not otherwise provide a way to detect the server version at runtime. If a newer driver is used with an older server, set the database version translator property manually.

Warning	The Phoenix driver does not have robust handling of time values. If your time values are normalized to use a date component of 1970-01-01, then the default handling will work correctly. If not, then the time column should be modeled as timestamp instead.

Cloudera Impala translator (impala)

Also see common JDBC Translators information.

The Cloudera Impala translator, known by the type name ***impala***, is for use with Cloudera Impala 1.2.1 or later.

Impala has limited support for data types. It is does not have native support for time/date/xml or LOBs. These limitations are reflected in the translator capabilities. A 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.

Do not use the `DatabaseTimeZone` translator property with the Impala translator.

Impala only supports EQUI join, so using any other joins types on its source tables will result in inefficient queries.

To write criteria based on partitioned columns, model them on the source table, but do not include them in selection columns.

Note	Impala Hive importer does not have concept of catalog or source schema, nor does it import keys, procedures, indexes, etc.

Impala specific importer properties

useDatabaseMetaData
Set to true to use the normal import logic with the option to import index information disabled. Defaults to false.

If the value of `useDatabaseMetaData` is false, the typical JDBC `DatabaseMetaData` calls are not used, so not all of the common JDBC importer properties are applicable to Impala. You may still use `excludeTables`, regardless.

Important	Some versions of Impala require the use of a `LIMIT` when performing an `ORDER BY`. If no default is configured in Impala, an exception can occur when a 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. For more information about Impala limit options, such as how to control what happens when the limit is exceeded, see the Cloudera documentation.

Note	If the Impala JDBC driver has problems processing `PreparedStatements` or parsing statements in general, try disabling `useBindVariables`. For more information, see https://issues.redhat.com/browse/TEIID-4610.

Db2 Translator (db2)

Also see common JDBC Translators information.

The Db2 translator, known by the type name ***db2***, is for use with IBM Db2 V8 or later, or IBM Db2 for i V5.4 or later.

Db2 execution properties

DB2ForI
Indicates that the the Db2 instance is Db2 for i. Defaults to `false`.

supportsCommonTableExpressions
Indicates that the Db2 instance supports common table expressions (CTEs). Defaults to `true`. Common table expression are not fully supported on some older versions of Db2, and on instances of Db2 that run in a conversion mode. If you encounter errors working with CTEs in these environments, set the CTE property to `false`.

Derby translator (derby)

Also see common JDBC Translators information.

The Derby translator, known by the type name ***derby***, is for use with Derby 10.1 or later.

Exasol translator (exasol)

Also see common JDBC Translators information.

The Exasol translator, known by the type name ***exasol***, is for use with Exasol version 6 or later.

Usage
The Exasol database has the NULL HIGH default ordering, whereas the Teiid engine works in the NULL LOW mode. As a result, depending on whether the ordering is pushed down to Exasol or done by the engine, you might observe NULLs at either the beginning or end of returned results. To enforce consistency, you can run Teiid with `org.teiid.pushdownDefaultNullOrder=true` to specify NULL LOW ordering. Enforcing NULL LOW ordering can result in decreased performance.

Greenplum Translator (greenplum)

Also see common JDBC Translators information.

The Greenplum translator, known by the type name ***greenplum***, is for use with the Greenplum database. This translator is an extension of the PostgreSQL translator, and inherits its options.

H2 Translator (h2)

Also see common JDBC Translators information.

The H2 Translator, known by the type name ***h2***, is for use with H2 version 1.1 or later.

Hive Translator (hive)

Also see common JDBC Translators information.

The Hive translator, known by the type name ***hive***, is for use with Hive v.10 and SparkSQL v1.0 and later.

Capabilities
Hive is compatible with a limited set of data types. It does not have native support for time/XML or large objects (LOBs). These limitations are reflected in the translator capabilities. Although a Teiid view can use these types, the transformation must specify the necessary conversions. Note that in those situations, evaluations are processed in Teiid engine.

Do not use the `DatabaseTimeZone` translator property with the Hive translator.

Hive only supports EQUI join, so using any other joins types on its source tables will result in inefficient queries.

To write criteria based on partitioned columns, model them on the source table, but do not include them in selection columns.

Note	The Hive importer does not have the concept of catalog or source schema, nor does it import keys, procedures, indexes, and so forth.

Import properties

trimColumnNames
For Hive 0.11.0 and later, the `DESCRIBE` command metadata is inappropriately returned with padding. Set this property to `true` to remove white space from column names. Defaults to `false` .

useDatabaseMetaData
For Hive 0.13.0 and later, the normal JDBC `DatabaseMetaData` facilities are sufficient to perform an import. Set to `true` to use the normal import logic with the option to import index information disabled. Defaults to `false` . When true, `trimColumnNames` has no effect. If it is set to false, the typical JDBC DatabaseMetaData calls are not used, so not all of the common JDBC importer properties are applicable to Hive. You can still use excludeTables anyway.

"Database Name"
When the database name used in Hive differs from `default` , the metadata retrieval and execution of queries does not work as expected in Teiid. The Hive JDBC driver seems to be implicitly connecting (tested with < 0.12) to "default" database, thus ignoring the database name mentioned on connection URL. You can work around this issue if you configure your connection source to send the command `use {database-name}` .

Teiid in WildFly environment set the following in data source configuration.

```
<new-connection-sql>use {database-name}</new-connection-sql>
```

This is fixed in version 0.13 and later of the Hive JDBC driver. For more information, see https://issues.apache.org/jira/browse/HIVE-4256.

Limitations
Empty tables might report their description without datatype information. To work around this problem when importing, you can exclude empty tables, or use the `useDatabaseMetaData` option.

HSQL Translator (hsql)

Also see common JDBC Translators information.

The HSQL Translator, known by the type name ***hsql***, is for use with HSQLDB 1.7 or later.

Informix translator (informix)

Also see common JDBC Translators information.

The Informix translator, known by the type name ***informix***, is for use with any Informix version.

Known issues

TEIID-3808
: The Informix driver's handling of timezone information is inconsistent, even if the `databaseTimezone` translator property is set. Verify that the Informix server and the application server are in the same time zone.

Ingres translators (ingres / ingres93)

Also see common JDBC Translators information.

You can use one of the following Ingres translators, depending on your Ingres version:

ingres
The Ingres translator, known by the type name ***ingres***, is for use with Ingres 2006 or later.

ingres93
The Ingres93 translator, known by the type name ***ingres93***, is for use with Ingres 9.3 or later.

Intersystems Caché translator (intersystems-cache)

Also see common JDBC Translators information.

The Intersystem Caché translator, known by the type name ***intersystems-cache***, is for use with Intersystems Caché Object database (relational aspects only).

JDBC ANSI translator (jdbc-ansi)

Also see common JDBC Translators information.

The JDBC ANSI translator, known by the type name ***jdbc-ansi***, works with most of the SQL constructs used in Teiid, except for row LIMIT/OFFSET and EXCEPT/INTERSECT. It translates source SQL into ANSI compliant syntax. This translator should be used when another more specific type is not available. If source exceptions arise due to the use of incompatible SQL constructs, then consider using the JDBC simple translator to further restrict capabilities, or create a custom translator. For more information, see the Custom Translator documentation in the Teiid community.

JDBC simple translator (jdbc-simple)

Also see common JDBC Translators information.

The JDBC Simple translator, known by the type name ***jdbc-simple***, is the same as the jdbc-ansi-translator, except that, to provide maximum compatibility, it does not handle most pushdown constructs.

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 Translators information.

access

The Microsoft Access translator known by the type name ***access*** is for use with Microsoft Access 2003 or later via the JDBC-ODBC bridge.

If you are using the default native metadata import, or the 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 might need to adjust these values if you use a different JDBC driver.

ucanaccess

The Microsoft Access translator known by the type name ***ucanaccess*** is for use with Microsoft Access 2003 or later via the UCanAccess driver.

Microsoft SQL Server translator (sqlserver)

Also see common JDBC translators information.

The Microsoft SQL Server translator, known by the type name ***sqlserver***, is for use with SQL Server 2000 or later. A SQL Server JDBC driver version 2.0 or later (or a compatible driver such as, JTDS 1.2 or later) should be used. The SQL Server `DatabaseVersion` property can be set to `2000`, `2005`, `2008`, or `2012`, but otherwise expects a standard version number, for example, `10.0`.

Sequences
With 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
: Specifies that use of the open source JTDS driver. 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)

Also see common JDBC translators information.

You can use the following translators with MySQL and MariaDB:

mysql
The MySQL translator, known by the type name ***mysql***, is for use with MySQL version 4.x or later. The translator also works with other compatible MySQL derivatives, such as MariaDB.

mysql5
The legacy MySQL5 translator, known by the type name ***mysql5***, was for use with MySQL version 5 or later. It has been incorporated into the mysql translator.

Usage
The MySQL translators expect the database or session to be using ANSI mode. If the database is not using ANSI mode, you can set ANSI mode on the pool by submitting the following initialization query:

```
set SESSION sql_mode = 'ANSI'
```

When data includes null timestamp values, Teiid generates the following conversion error: `0000-00-00 00:00:00 cannot be converted to a timestamp`. To avoid error, if you expect data with null timestamp values, set the connection property `zeroDateTimeBehavior=convertToNull`.

Warning	If you must retrieve large result sets, consider setting the connection property `useCursorFetch=true`. Otherwise MySQL will fully fetch result sets into memory on the 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, change the BOOLEAN columns in the affected source to have a native type of "TINYINT(1)" rather than BIT so that the translator can appropriately handle the Boolean conversion.

Netezza translator (netezza)

Also see common JDBC translators information.

The Netezza translator, known by the type name ***netezza***, is for use with any version of the IBM Netezza appliance.

Usage
The current vendor-supplied JDBC driver for Netezza performs poorly with single transactional updates. It is best to perform batched updates whenever possible.

Execution properties
Netezza-specific execution properties:

SqlExtensionsInstalled
Indicates that SQL extensions, including the ability to process Netezza `REGEXP_LIKE` functions, are installed. All other REGEXP functions are then available as pushdown functions. Defaults to `false` .

Oracle translator (oracle)

Also see common JDBC translators information.

The Oracle translator, known by the type name ***oracle***, is for use with Oracle Database 9i or later.

Note	The Oracle-provided JDBC driver uses large amounts of memory. Because the driver caches a high volume of data in the buffer, problems can occur on computers that lack sufficient memory allocation. For more information, see the following resources: • Teiid issue. • Oracle whitepaper.

Importer properties

useGeometryType
: Use the 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
: You can use sequences with the Oracle translator. You can model a sequence as a table with a name in source of DUAL, and setting column names in the source set to `<sequence name>.[nextval|currval]`

With Teiid 10.0+, you can import sequences automatically.

For more information, see *Importer properties* in JDBC translators. 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.

For information about representing `currval` and `nextval` as source functions, see DDL metadata for schema objects

8.5 Example: Sequence native query

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

You can also use a sequence as the default value for insert columns by setting the column to autoincrement, and setting the name in source to `<element name>:SEQUENCE=<sequence name>.<sequence value>` .

Rownum

A `rownum` column can also be added to any Oracle physical table to enable use of the rownum pseudo-column. A rownum column should have a name in source of `rownum`. These rownum columns do not have the same semantics as the Oracle rownum construct so care must be taken in their usage.

Out parameter result set
Out parameters for procedures may also be used to return a result set, if this is not represented correctly by the automatic import you need to manually create a result set and represent the output parameter with native type `REF CURSOR`.

DDL for out parameter result set

```
create foreign procedure proc (in x integer, out y object options (native_type 'REF CURSOR'))
returns table (a integer, b string)
```

Geospatial functions
You can use the following geospatial functions with the translator for Oracle:

Relate = sdo_relate

```
CREATE FOREIGN FUNCTION sdo_relate (arg1 string,  arg2 string,  arg3 string) RETURNS string;
CREATE FOREIGN FUNCTION sdo_relate (arg1 Object,  arg2 Object,  arg3 string) RETURNS string;
CREATE FOREIGN FUNCTION sdo_relate (arg1 string,  arg2 Object,  arg3 string) RETURNS string;
CREATE FOREIGN FUNCTION sdo_relate (arg1 Object,  arg2 string,  arg3 string) RETURNS string;
```

Nearest_Neighbor = sdo_nn

```
CREATE FOREIGN FUNCTION sdo_nn (arg1 string,  arg2 Object,  arg3 string,  arg4 integer) RETURNS string;
CREATE FOREIGN FUNCTION sdo_nn (arg1 Object,  arg2 Object,  arg3 string,  arg4 integer) RETURNS string;
CREATE FOREIGN FUNCTION sdo_nn (arg1 Object,  arg2 string,  arg3 string,  arg4 integer) RETURNS string;
```

Within_Distance = sdo_within_distance

```
CREATE FOREIGN FUNCTION sdo_within_distance (arg1 Object,  arg2 Object,  arg3 string) RETURNS string;
CREATE FOREIGN FUNCTION sdo_within_distance (arg1 string,  arg2 Object,  arg3 string) RETURNS string;
CREATE FOREIGN FUNCTION sdo_within_distance (arg1 Object,  arg2 string,  arg3 string) RETURNS string;
```

Nearest_Neigher_Distance = sdo_nn_distance

```
CREATE FOREIGN FUNCTION sdo_nn_distance (arg integer) RETURNS integer;
```

Filter = sdo_filter

```
CREATE FOREIGN FUNCTION sdo_filter (arg1 Object,  arg2 string,  arg3 string) RETURNS string;
CREATE FOREIGN FUNCTION sdo_filter (arg1 Object,  arg2 Object,  arg3 string) RETURNS string;
CREATE FOREIGN FUNCTION sdo_filter (arg1 string,  arg2 object,  arg3 string) RETURNS string;
```

Pushdown functions
Depending on the Oracle version, the Oracle translator, registers the following non-geospatial pushdown functions with the engine:

TRUNC
Both numeric and timestamp versions.

LISTAGG
Requires the 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 make the following changes:

- Use client driver version 11, or later.
- Place the `xdb.jar` and `xmlparserv2.jar` files in the classpath.
- Set the system property `oracle.jdbc.getObjectReturnsXMLType="false"` .

  For more information, see the Oracle documentation.

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.

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.

JCA Resource Adapter

The resource adapter for this translator is provided through OSISoft PI Data Sources. Refer to Admin Guide for configuration.

PostgreSQL translator (postgresql)

Also see common JDBC translators information.

The PostgreSQL translator, known by the type name ***postgresql***, is for use with the following PostgreSQL client and server versions: * Client — 8.0 or later * Server — 7.1 or later.

Execution properties
PostgreSQL-specific execution properties:

PostGisVersion
Indicates the PostGIS version in use. Defaults to 0, which means that PostGIS is not installed. Will be set automatically if the database version is not set.

ProjSupported
Boolean that indicates if the PostGis version supports PROJ coordinate transformation software. Will be set automatically if the database version is not set.

Note	Some driver versions of PostgreSQL will not associate columns to "INDEX" type tables. The current version of Teiid omits such tables automatically. Older versions of Teiid may need the importer.tableType property or other filtering set.

PrestoDB translator (prestodb)

Also see common JDBC translators information.

The PrestoDB translator, known by the type name ***prestodb***, exposes querying functionality to Presto data sources. In data integration respect, PrestoDB has capabilities that are similar to 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. Teiid provides more complete querying capabilities and many enterprise features.

Capabilities
You can use the PrestoDB translator only with `SELECT` statements. The translator provides a restricted set of capabilities.

Because PrestoDB exposes a relational model, Teiid can use it as it does other RDBMS sources, such as Oracle, Db2, and so forth. For information about configuring PrestoDB, see the Presto documentation.

Tip	In SQL JOIN operations, PrestoDB does not support multiple `ORDER BY` columns well. If errors occur during JOIN operations that involve more than one `ORDER BY` column, set the translator property `supportsOrderBy` to disable the use of the `ORDER BY` clause.

Note	Some versions of Presto generate errors when you include null values in subqueries.

Tip	PrestoDB does not support transactions. To overcome issues caused by this limitation, define the data source as non-transactional.

<table><tr><td>Note</td><td>By default, every catalog in PrestoDB has an <code>information_schema</code>. If you have to configure multiple catalogs, duplicate table errors can cause deployment of a virtual database to fail. To prevent duplicate table errors, use import options to filter the schemas.

If you want to configure multiple Presto catalogs, set one of the following import options to filter the schemas and tables in the source:
• Set <code>catalog</code> to a specific catalog name to match the name of the source catalog in Presto.
• Set <code>schemaName</code> to a regular expression to filter schemas by matching result.
• Set <code>excludeTables</code> to a regular expression to filter tables by matching results.</td></tr></table>

Redshift translator (redshift)

Also see common JDBC translators information.

The Redshift translator, known by the type name ***redshift***, is for use with the Amazon Redshift database. This translator is an extension of the PostgreSQL translator and inherits its options.

SAP HANA translator (hana)

Also see common JDBC translators information.

The SAP HANA translator, known by the name of ***hana***, is for use with SAP HANA.

Known issues

TEIID-3805
The pushdown of the SUBSTRING function is inconsistent with the 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 translators information.

The SAP IQ translator, known by the type name ***sap-iq***, is for use with SAP IQ version 15.1 or later. The translator name ***sybaseiq*** has been deprecated.

Sybase translator (sybase)

Also see common JDBC Translators information.

The Sybase translator, known by the type name ***sybase***, is for use with SAP ASE (Adaptive Server Enterprise), formerly known as Sybase SQL Server, version 12.5 or later.

If you use the default native import, you can avoid exceptions during the retrieval of system table information, if you specify import properties. If errors occur when retrieving table information, specify a `schemaName` or `schemaPattern`, or use `excludeTables` to exclude system tables. For more information about using import properties, see *Importer properties* in JDBC translators.

If the name in the source metadata contains quoted identifiers (such as reserved words, or words that contain characters that would not otherwise be allowed), and you are using a jConnect Sybase driver, you must first configure the connection pool to enable **quoted_identifier**:

Example: Driver URL with SQLINITSTRING

```
jdbc:sybase:Tds:host.at.some.domain:5000/db_name?SQLINITSTRING=set quoted_identifier on
```

Important	If you are using a jConnect Sybase driver and will target the source for dependent joins, set the `JCONNECT_VERSION` to `6` or later to increase the number of values that the translator can send. If you do not set the `JCONNECT_VERSION`, an exception occurs with statements that have more than 481 bind values.

Example: Driver URL with JCONNECT_VERSION

```
jdbc:sybase:Tds:host.at.some.domain:5000/db_name?SQLINITSTRING=set quoted_identifier on&JCONNECT_VERSION=6
```

Execution properties specific to Sybase

JtdsDriver_
: Indicates that the open source JTDS driver is being used. Defaults to false.

Teiid translator (teiid)

Also see common JDBC translators information.

Use the Teiid translator, known by the type name ***teiid***, when creating a virtual database from a Teiid data source.

Teradata translator (teradata)

Also see common JDBC translators information.

The Teradata translator, known by the type name ***teradata***, is for use with Teradata Database V2R5.1 or later.

By default, Teradata driver version 15, adjusts date, time, and timestamp values to match the 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 translators 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 compatible with 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 they can process only equality predicates. 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 Attributes

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 capabilities are not 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.

Arrays

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>
```

For more ways to create the connector see LDAP Data Sources.

The LDAP translator cannot derive the metadata, the user needs 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 works with all SQL constructs and returns default results, with some configurable behavior.

Table 1. 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 which is to be mimicked.	na

You can also use the Loopback translator to mimic how a real source query would be formed for a given translator (although loopback will still return dummy data that might not be useful for your situation). To enable this behavior, set the `DelegateName` property to the name of the translator that you want to mimic. For example, to disable all capabilities, set the `DelegateName` property to `jdbc-simple`.

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 a Microsoft Excel document. This translator provides an easy way read a Excel spreadsheet and provide the contents of the spreadsheet in a tabular form that can be integrated with other sources in Teiid.

Note	This translator works on all platforms, including Windows and Linux. The translator uses Apache POI libraries to access the Excel documents which are platform independent.

Translation mapping

The following table describes how Excel translator interprets the data in Excel document into relational terms.

Excel Term	Relational term
Workbook	schema
Sheet	Table
Row	Row of data
Cell	Column Definition or Data of a column

The Excel translator provides a "source metadata" feature, where for a given Excel workbook, it can introspect and build the schema based on the worksheets that are defined within it. There are options available to detect header columns and data columns in a worksheet to define the correct metadata of a table.

DDL example

The following example shows how to expose an Excel spreadsheet in a virtual database.

```
CREATE DATABASE excelvdb;
USE DATABASE excelvdb;
CREATE SERVER connector FOREIGN DATA WRAPPER excel OPTIONS ("resource-name" 'java:/fileDS');
CREATE SCHEMA excel SERVER connector;
SET SCHEMA excel;
IMPORT FROM SERVER connector INTO excel OPTIONS (
    "importer.headerRowNumber" '1',
    "importer.ExcelFileName" 'names.xls');
```

As an XML VDB:

```
<?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 the preceding example represents the connection to the Excel document.

Headers in document

If the Excel document contains headers, you can guide the import process to select the cell headers as the column names in the table creation process. For information about defining import properties, see the following table, and also see *Importer Properties* in JDBC translators.

Import properties

Import properties guide the schema generation part during the deployment of the VDB. This can be used in a native import.

Property Name	Description	Default
importer.excelFileName	Defines the name of the Excel Document to import metadata. This can be defined as a file pattern (*.xls), however when defined as pattern all files must be of same format, and the translator will choose an arbitrary file to import metadata from. Use file patterns to read data from multiple Excel documents in the same directory. In the case of a single file, specify the absolute name.	Required
importer.headerRowNumber	Defines the cell header information to be used as column names.	Optional. Default is first data row of sheet
importer.dataRowNumber	Defines the row number where the data rows start.	Optional. Default is first data row of sheet.

To enable information in the Excel spreadsheet to be interpreted correctly, it is best to define all the preceding importer properties.

Note	Purely numerical cells in a column contain containing mixed types will have a string form matching their decimal representation, thus integral values will have `.0` appended. If you need the exact text representation, then the cell must be a string value. You can force a string value by preceding the numeric text of a cell with a single quote (`'`), or a single space.

Translator extension properties

- Excel specific execution properties

  FormatStrings

  Format non-string cell values in a string column according to the worksheet format. Defaults to false.

- Metadata extension properties

  Properties that are defined on schema artifacts, such as Table, Column, Procedure and so forth. These properties describe how the translator interacts with or interprets source systems. All the properties are defined with the following namespace: `"http://www.teiid.org/translator/excel/2014[http://www.teiid.org/translator/excel/2014\]"` , which also has a recognized alias `teiid_excel` .

Property Name	Schema item property belongs to	Description	Mandatory
FILE	Table	Defines Excel Document name or name pattern (*.xls). File pattern can be used to read data from multiple files.	Yes
FIRST_DATA_ROW_NUMBER	Table	Defines the row number where records start in the sheet (applies to every sheet).	Optional
CELL_NUMBER	Column of Table	Defines cell number to use for reading data of particular column.	Yes

The following example shows a table that is defined by using the extension metadata properties.

```
CREATE DATABASE excelvdb;
USE DATABASE excelvdb;
CREATE SERVER connector FOREIGN DATA WRAPPER excel OPTIONS ("resource-name" 'java:/fileDS');
CREATE SCHEMA excel SERVER connector;
SET SCHEMA excel;
CREATE FOREIGN TABLE Person (
                ROW_ID integer OPTIONS (SEARCHABLE 'All_Except_Like', "teiid_excel:CELL_NUMBER" 'ROW_ID'),
                FirstName string OPTIONS (SEARCHABLE 'Unsearchable', "teiid_excel:CELL_NUMBER" '1'),
                LastName string OPTIONS (SEARCHABLE 'Unsearchable', "teiid_excel:CELL_NUMBER" '2'),
                Age integer OPTIONS (SEARCHABLE 'Unsearchable', "teiid_excel:CELL_NUMBER" '3'),
                CONSTRAINT PK0 PRIMARY KEY(ROW_ID)
             ) OPTIONS ("NAMEINSOURCE" 'Sheet1',"teiid_excel:FILE" 'names.xlsx', "teiid_excel:FIRST_DATA_ROW_NU
MBER" '2')
```

As an XML VDB:

```
<?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>
```

Extended capabilities using ROW_ID column

If you define a column that has extension metadata property `CELL_NUMBER` with value `ROW_ID`, then that column value contains the row information from Excel document. You can mark this column as Primary Key. You can use this column in `SELECT` statements with a restrictive set of capabilities including: comparison predicates, `IN` predicates and `LIMIT`. All other columns cannot be used as predicates in a query.

Tip	Importing source metadata is not the only way to create the schema of an Excel document. You can also create a source table manually, and then add the extension properties that you need to create a fully functional model. Metadata imports result in schema models similar to the one in the preceding example.

The Excel translator processes updates with the following limitations:

- The `ROW_ID` cannot be directly modified or used as an insert value.
- UPDATE and INSERT values must be literals.
- UPDATEs are not transactional. That is, the write lock is held while the file is written, but not throughout the entire update. As a result, it is possible for one update to overwrite another.

The `ROW_ID` of an inserted row can be returned as a generated key.

JCA resource adapter

See File Data Source, the FTP Data Source and the Admin Guide in general for configuration information.

Note	Native queries This feature is not applicable for the Excel translator.

Note	Direct query procedure This feature is not applicable for the 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 provides for a full range of SELECT, INSERT, UPDATE and DELETE calls.

MongoDB is a document based "schema-less" database with it own query language. It does not map perfectly with relational concepts or the SQL query language. More and more systems are using NOSQL stores such as MongoDB to improve scalability and performance. For example, applications like storing audit logs, or managing web site data, are well-suited to MongoDB, and do not require the structure of relational databases. MongoDB uses JSON documents as its primary storage unit, and those documents can have additional embedded documents inside the parent document. By using embedded documents, MongoDB co-locates related information to achieve de-normalization that typically requires either duplicate data or joins to achieve querying in a relational database.

For MongoDB to work with Teiid the challenge for the MongoDB translator is to design a MongoDB store that can achieve the balance between relational and document based storage. The advantages of "schema-less" design are great at development time. But "schema-less" design can pose problems during migration between application versions, and when querying data, and making effective use of the returned information.

Since it is hard and may be impossible in certain situations to derive a schema based on existing the MongoDB collection(s), Teiid approaches the problem in reverse compared to other translators. When working with MongoDB, Teiid requires you to define the MongoDB schema upfront, by using Teiid metadata. Because Teiid only allows relational schema as its metadata, you must define your MongoDB schema in relational terms, using tables, procedures, and functions. For the purposes of MongoDB, the Teiid metadata has been extended to provide extension properties that can be defined on a table to convert it into a MongoDB based document. These extension properties let you define how a MongoDB document is structured and stored. Based on the relationships (primary-key, foreign-key) that are defined on a table, and their cardinality (ONE-to-ONE, ONE-to-MANY, MANY-to-ONE), relations between tables are mapped such that related information can be embedded along with the parent document for co-location (as mentioned earlier in this topic). Thus, a relational schema-based design, but document-based storage in MongoDB.

Who is the primary audience for the MongoDB translator?
The above may not satisfy every user’s needs. The document structure in MongoDB can be more complex than what Teiid can currently define. We hope this will eventually catch up in future versions of Teiid. This is currently designed for:

- Users who are using relational databases and would like to move/migrate their data to MongoDB to take advantage of scaling and performance without modifying end user applications that they currently run.
- Users who are seasoned SQL developers, but do not have experience with MongoDB. This provides a low barrier of entry compared to using MongoDB directly as an application developer.
- Users who want to integrate MongoDB-based data with data from other enterprise data sources.

Usage
The name of the translator to use in a virtual database DDL is "*mongodb*". For example:

```
CREATE DATABASE nothwind;
USE DATABASE nothwind;
CREATE SERVER local FOREIGN DATA WRAPPER mongodb OPTIONS ("resource-name" 'java:/mongoDS');
CREATE SCHEMA northwind SERVER local;

SET SCHEMA northwind;
IMPORT FROM SERVER local INTO northwind;
```

Or as an xml vdb:

```
<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 database. For that purpose, Teiid has a JCA adapter that provides a connection to MongoDB using the MongoDB Java Driver. To define such a 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>
```

For information about more ways to create the connector see *MongoDB data sources* in the Administrator’s Guide.

The MongoDB translator can derive the metadata based on existing document collections in some scenarios. However, when working with complex documents the interpretation of metadata can be inaccurate. In such cases, you must define the metadata. For example, you can define a schema using DDL, as shown in the following example:

```
<vdb name="nothwind" version="1">
    <model name="northwind">
        <source name="local" translator-name="mongodb" connection-jndi-name="java:/mongoDS"/>
            <metadata type="DDL"><![CDATA[
                CREATE FOREIGN TABLE  Customer (
                    customer_id integer,
                    FirstName varchar(25),
                    LastName varchar(25)
                ) OPTIONS(UPDATABLE 'TRUE');
            ]]> </metadata>
    </model>
<vdb>
```

When the following INSERT operation is executed against a table using Teiid, the MongoDB translator creates a document in the MongoDB:

```
    INSERT INTO Customer(customer_id, FirstName, LastName) VALUES (1, 'John', 'Doe');
```

```
{
  _id: ObjectID("509a8fb2f3f4948bd2f983a0"),
  customer_id: 1,
  FirstName: "John",
  LastName: "Doe"
}
```

If a PRIMARY KEY is defined on the table, then that column name is automatically used as `"_id"` field in the MongoDB collection, and then the document structure is stored in the MongoDB, as shown in the following examples:

```
    CREATE FOREIGN TABLE  Customer (
        customer_id integer PRIMARY KEY,
        FirstName varchar(25),
        LastName varchar(25)
    ) OPTIONS(UPDATABLE 'TRUE');
```

```
{
  _id: 1,
  FirstName: "John",
  LastName: "Doe"
}
```

If you defined the composite PRIMARY KEY on Customer table, the document structure that results is shown in the following example:

```
    CREATE FOREIGN TABLE  Customer (
        customer_id integer,
        FirstName varchar(25),
        LastName varchar(25),
        PRIMARY KEY (FirstName, LastName)
    ) OPTIONS(UPDATABLE 'TRUE');
```

```
{
  _id: {
         FirstName: "John",
         LastName:  "Doe"
       },
  customer_id: 1,
}
```

Data types

The MongoDB translator provides automatic mapping of Teiid data types into MongoDB data types, including BLOBS, CLOBS and XML. The LOB mapping is based on GridFS in MongoDB. Arrays are in the following form:

```
{
  _id: 1,
  FirstName: "John",
  LastName: "Doe"
  Score: [89, "ninety", 91.0]
}
```

Users can get individual items in the array using the function `array_get` , or can transform the array into tabular structure using ARRAYTABLE.

Note	Note that even though embedded documents can also be in arrays, the handling of embedded documents is different from array with scalar values.

Note	The translator does not work with regular Expressions, MongoDB::Code, MongoDB::MinKey, MongoDB::MaxKey, and MongoDB::OID.

Note	In documents that contain values of mixed types for the same key, you must mark the column as unsearchable, or MongoDB will not correctly match predicates against the column. A key is used as a mixed type of it is represented as a string value in one document, and an integer in another. For more information, see the `importer.sampleSize property` in the following table.

Importer Properties

Importer properties define the behavior of the translator during the metadata import from the physical source.

Importer Properties

Name	Description	Default
excludeTables	Regular expression to exclude the tables from import.	null
includeTables	Regular expression to include the tables from import.	null
sampleSize	Number of documents to sample to determine the structure. If documents have different fields, or fields with different types, this should be greater than 1.	1
fullEmbeddedNames	Whether to prefix embedded table names with their parents, e.g. parent_embedded. If false the name of the table will just be the name of the field - which may lead to conflicts with existing tables or other embedded tables.	false

MongoDB metadata extension properties for building complex documents

Using the preceding DDL, or any other metadata facility, you can map a table in a relational store into a document in MongoDB. However, to make effective use of MongoDB, you must be able to build complex documents that can co-locate related information, so that data can queried in a single MongoDB query. Unlike a relational database, you cannot run join operations in MongoDB. As as a result, unless you can build complex documents, you would have to issue multiple queries to retrieve data and then join it manually. The power of MongoDB comes from its "embedded" documents, its support for complex data types, such as arrays, and its use of an aggregation framework to query them. This translator provides a way to achieve the goals.

When you do not define the complex embedded documents in MongoDB, 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. For more information about Teiid schema metadata, see [ddl-metadata-for-schema-objects]. You can use the following metadata properties in your DDL:

teiid_mongo:EMBEDDABLE

Means that data defined in this table is allowed to be included as an "embeddable" document in **any** parent document. The parent document is referenced by the foreign key relationships. In this scenario, 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 A given table can contain either the `teiid_mongo:EMBEDDABLE` property or the `teiid_mongo:MERGE` property defining the type of nesting in MongoDB. You cannot use both properties within one table.

ONE-2-ONE Mapping

If your current DDL structure representing ONE-2-ONE relationship is like

```
    CREATE FOREIGN TABLE  Customer (
        CustomerId integer PRIMARY KEY,
        FirstName varchar(25),
        LastName varchar(25)
    ) OPTIONS(UPDATABLE 'TRUE');

    CREATE FOREIGN TABLE Address (
        CustomerId integer,
        Street varchar(50),
        City varchar(25),
        State varchar(25),
        Zipcode varchar(6),
        FOREIGN KEY (CustomerId) REFERENCES Customer (CustomerId)
     ) OPTIONS(UPDATABLE 'TRUE');
```

By default, this will produce two different collections in MongoDB, like with sample data it will look like

```
Customer
{
  _id: 1,
  FirstName: "John",
  LastName: "Doe"
}

Address
{
  _id: ObjectID("..."),
   CustomerId: 1,
   Street: "123 Lane"
   City: "New York",
   State: "NY"
   Zipcode: "12345"
}
```

You can enhance the storage in MongoDB to a single collection by using `teiid_mongo:MERGE` extension property on the table's OPTIONS clause.

```
    CREATE FOREIGN TABLE  Customer (
        CustomerId integer PRIMARY KEY,
        FirstName varchar(25),
        LastName varchar(25)
    ) OPTIONS(UPDATABLE 'TRUE');

    CREATE FOREIGN TABLE Address (
        CustomerId integer PRIMARY KEY,
        Street varchar(50),
        City varchar(25),
        State varchar(25),
        Zipcode varchar(6),
        FOREIGN KEY (CustomerId) REFERENCES Customer (CustomerId)
```

```
    ) OPTIONS(UPDATABLE 'TRUE', "teiid_mongo:MERGE" 'Customer');
```

this will produce single collection in MongoDB, like

```
Customer
{
  _id: 1,
  FirstName: "John",
  LastName: "Doe",
  Address:
     {
        Street: "123 Lane",
        City: "New York",
        State: "NY",
        Zipcode: "12345"
     }
}
```

With the above both tables are merged into a single collection that can be queried together using the JOIN clause in the SQL command. Since the existence of child/additional record has no meaning with out parent table using the "*teiid_mongo:MERGE*" extension property is right choice in this situation.

Note	The Foreign Key defined on a child table must refer to Primary Keys on both the parent and child tables to form a One-2-One relationship.

ONE-2-MANY Mapping.
Typically there can be more than two (2) tables involved in this relationship. If MANY side is only associated **single** table, then use `teiid_mongo:MERGE` property on MANY side of table and define ONE as the parent. If associated with more than single table then use `teiid_mongo:EMBEDDABLE` .

For example, if you define a virtual database as in the following DDL:

```
    CREATE FOREIGN TABLE  Customer (
        CustomerId integer PRIMARY KEY,
        FirstName varchar(25),
        LastName varchar(25)
    ) OPTIONS(UPDATABLE 'TRUE');

    CREATE FOREIGN TABLE  Order (
        OrderID integer PRIMARY KEY,
        CustomerId integer,
        OrderDate date,
        Status integer,
        FOREIGN KEY (CustomerId) REFERENCES Customer (CustomerId)
    ) OPTIONS(UPDATABLE 'TRUE');
```

then a Single Customer can have MANY Orders. There are two options to define the how we store the MongoDB document. If in your schema, the Customer table's CustomerId is **only** referenced in Order table (i.e. Customer information used for only Order purposes), you can use

```
    CREATE FOREIGN TABLE  Customer (
        CustomerId integer PRIMARY KEY,
        FirstName varchar(25),
        LastName varchar(25)
    ) OPTIONS(UPDATABLE 'TRUE');

    CREATE FOREIGN TABLE  Order (
        OrderID integer PRIMARY KEY,
        CustomerId integer,
        OrderDate date,
        Status integer,
```

```
        FOREIGN KEY (CustomerId) REFERENCES Customer (CustomerId)
    ) OPTIONS(UPDATABLE 'TRUE', "teiid_mongo:MERGE" 'Customer');
```

that will produce a single document for Customer table like

```
{
  _id: 1,
  FirstName: "John",
  LastName: "Doe",
  Order:
  [
     {
       _id: 100,
        OrderDate: ISODate("2000-01-01T06:00:00Z")
        Status: 2
     },
     {
       _id: 101,
        OrderDate: ISODate("2001-03-06T06:00:00Z")
        Status: 5
     }
     ...
   ]
}
```

If Customer table is referenced in more tables other than Order table, then use "teiid_mongo:EMBEDDABLE" property

```
    CREATE FOREIGN TABLE Customer (
        CustomerId integer PRIMARY KEY,
        FirstName varchar(25),
        LastName varchar(25)
    ) OPTIONS(UPDATABLE 'TRUE', "teiid_mongo:EMBEDDABLE" 'TRUE');

    CREATE FOREIGN TABLE Order (
        OrderID integer PRIMARY KEY,
        CustomerId integer,
        OrderDate date,
        Status integer,
        FOREIGN KEY (CustomerId) REFERENCES Customer (CustomerId)
    ) OPTIONS(UPDATABLE 'TRUE');

    CREATE FOREIGN TABLE Comments (
        CommentID integer PRIMARY KEY,
        CustomerId integer,
        Comment varchar(140),
        FOREIGN KEY (CustomerId) REFERENCES Customer (CustomerId)
    ) OPTIONS(UPDATABLE 'TRUE');
```

This creates three different collections in MongoDB.

```
Customer
{
  _id: 1,
  FirstName: "John",
  LastName: "Doe"
}

Order
{
  _id: 100,
  CustomerId: 1,
  OrderDate: ISODate("2000-01-01T06:00:00Z")
  Status: 2
  Customer:
```

```
    {
      FirstName: "John",
      LastName: "Doe"
    }
}

Comment
{
  _id: 12,
  CustomerId: 1,
  Comment: "This works!!!"
  Customer:
    {
      FirstName: "John",
      LastName: "Doe"
    }
}
```

Here as you can see the Customer table contents are embedded along with other table's data where they were referenced. This creates duplicated data where multiple of these embedded documents are managed automatically in the MongoDB translator.

Note	All the SELECT, INSERT, DELETE operations that are generated against the tables with "teiid_mongo:EMBEDDABLE" property are atomic, except for UPDATES, as there can be multiple operations involved to update all the copies. Since there are no transactions in MongoDB, 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

- Nested embedding of documents is limited due to capabilities of handling nested arrays is limited in the MongoDB. Nesting of "EMBEDDABLE" property with multiple levels is OK, however more than two levels with MERGE is not recommended. Also, you need to be caution about not exceeding the document size of 16 MB for single row, so deep nesting is not recommended.
- JOINS between related tables, MUST use either the "EMBEDDABLE" or "MERGE" properties, otherwise the query will result in error. In order for Teiid to correctly plan and work with JOINS, in the case that any two tables are **NOT** embedded in each other, use *allow-joins=false* property on the Foreign Key that represents the relation. For example:

```
CREATE FOREIGN TABLE  Customer (
    CustomerId integer PRIMARY KEY,
    FirstName varchar(25),
    LastName varchar(25)
) OPTIONS(UPDATABLE 'TRUE');

CREATE FOREIGN TABLE  Order (
    OrderID integer PRIMARY KEY,
    CustomerId integer,
    OrderDate date,
    Status integer,
    FOREIGN KEY (CustomerId) REFERENCES Customer (CustomerId) OPTIONS (allow-join 'FALSE')
) OPTIONS(UPDATABLE 'TRUE');
```

with the example above, 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 functions
MongoDB translator enables you to use geo spatial query operators in the "WHERE" clause, when the data is stored in the GeoJSon format in the MongoDB Document. The following functions are available:

```
CREATE FOREIGN FUNCTION geoIntersects (columnRef string,  type string, coordinates double[][]) RETURNS boolean;
CREATE FOREIGN FUNCTION geoWithin (ccolumnRef string,  type string, coordinates double[][]) RETURNS boolean;
CREATE FOREIGN FUNCTION near (ccolumnRef string,  coordinates double[], maxdistance integer) RETURNS boolean;
CREATE FOREIGN FUNCTION nearSphere (ccolumnRef string, coordinates double[], maxdistance integer) RETURNS boolean;
CREATE FOREIGN FUNCTION geoPolygonIntersects (ref string, north double, east double, west double, south double) RETURNS boolean;
CREATE FOREIGN FUNCTION geoPolygonWithin (ref string, north double, east double, west double, south double) RETURNS boolean;
```

a sample query looks like

```
SELECT loc FROM maps where mongo.geoWithin(loc, 'LineString', ((cast(1.0 as double), cast(2.0 as double)), (cast(1.0 as double), cast(2.0 as double))))
```

Same functions using built-in Geometry type (the versions of the functions in the preceding list will be deprecated and removed in future versions)

```
CREATE FOREIGN FUNCTION geoIntersects (columnRef string,  geo geometry) RETURNS boolean;
CREATE FOREIGN FUNCTION geoWithin (ccolumnRef string,  geo geometry) RETURNS boolean;
CREATE FOREIGN FUNCTION near (ccolumnRef string, geo geometry, maxdistance integer) RETURNS boolean;
CREATE FOREIGN FUNCTION nearSphere (ccolumnRef string, geo geometry, maxdistance integer) RETURNS boolean;
CREATE FOREIGN FUNCTION geoPolygonIntersects (ref string, geo geometry) RETURNS
```

```
boolean;
CREATE FOREIGN FUNCTION geoPolygonWithin (ref string, geo geometry) RETURNS
boolean;
```

a sample query looks like

```
SELECT loc FROM maps where mongo.geoWithin(loc, ST_GeomFromGeoJSON('{"coordinates":[[1,2],[3,4]],"type":"Polygo
n"}'))
```

There are various "st_geom.." methods are available in the Geo Spatial function library in Teiid.

Capabilities
MongoDB translator is designed on top of the MongoDB aggregation framework. You must use a MongoDB version that the aggregation framework. Apart from SELECT queries, the MongoDB translator also works with INSERT, UPDATE and DELETE queries.

You can use the MongoDB translator with the following functions:

- Grouping.
- Matching.
- Sorting.
- Filtering.
- Limits.
- Working with LOBs stored in GridFS.
- Composite primary and foreign keys.

Native queries
MongoDB source procedures may be created using the `teiid_rel:native-query` extension. For more information, see *Parameterizable native queries* in Translators. The procedure will invoke the native-query similar to a direct procedure call with the benefits that the query is predetermined and that result column types are known, rather than requiring the use of ARRAYTABLE or similar functionality.

Direct query procedure
This feature is turned off by default because of the security risk this exposes to execute any command against the source. To enable direct query procedures, set the execution property called `SupportsDirectQueryProcedure` to `true`. For more information, see *Override the execution properties* in [as_translators].

By default the name of the procedure that executes the queries directly is called **native**. For information about how to change the default name, see *Override the execution properties* in [as_translators].

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 web services resource adapter for making web service calls. This translator is an extension of the *Web services translator*.

What is OData?
The Open Data Protocol (OData) web protocol is for querying and updating data that provides a way to unlock your data and free it from silos that exist in applications today. OData does this by applying and building upon Web technologies such as HTTP, Atom Publishing Protocol (AtomPub) and JSON to provide access to information from a variety of applications, services, and stores. OData is being used to expose and access information from a variety of sources including, but not limited to, relational databases, file systems, content management systems and traditional Web sites.

Using this specification from the OASIS group, with help from the OData4J framework, Teiid maps OData entities into relational schema. Teiid can read CSDL (Conceptual Schema Definition Language) from a provided OData endpoint, and convert the OData schema into a relational schema. The following table shows the mapping selections in the OData translator from a CSDL document.

OData	Mapped to relational entity
EntitySet	Table
FunctionImport	Procedure
AssociationSet	Foreign keys on the table*
ComplexType	ignored**

- A many-to-many association will result in a link table that can not be selected from, but can be used for join purposes.
    - When used in functions, an implicit table is exposed. When used to define a embedded table, all the columns will be in-lined.

All CRUD operations will be appropriately mapped to the resulting entity based on the SQL submitted to the OData translator.

1. Usage

Usage of a OData source is similar to that of a JDBC translator. The metadata import is provided through the translator, once the metadata is imported from the source system and exposed in relational terms, then this source can be queried as if the EntitySets and Function Imports were local to the Teiid system.

Table 1. Execution properties

Name	Description	Default
DatabaseTimeZone	The time zone of the database. Used when fetchings date, time, or timestamp values.	The system default time zone
SupportsOdataCount	Enables the use of the `$count` option in system queries.	true
SupportsOdataFilter	Enables the use of the `$filter` option in system queries.	true
SupportsOdataOrderBy	Enables the use of the `$orderby` option in system queries.	true

SupportsOdataSkip	Enables the use of the `$skip` option in system queries.	true
SupportsOdataTop	Enables the use of the `$top` option in system queries.	true

Table 2. **Importer Properties**

Name	Description	Default
schemaNamespace	Namespace of the schema to import.	null
entityContainer	Entity Container Name to import.	default container

Example: Importer settings to import only tables and views from NetflixCatalog

```
<property name="importer.schemaNamespace" value="System.Data.Objects"/>
<property name="importer.entityContainer" value="NetflixCatalog"/>
```

Note	OData Server is not fully compatible The OData server that you connect to might not fully implement the entire OData specification. If the server's OData implementation does not support a feature, set "execution properties" to turn off the corresponding capability, so that Teiid will not push down invalid queries to the translator.

For example, to turn off `$filter` , add following to your vdb.xml

```
    <translator name="odata-override" type="odata">
    <property name="SupportsOdataFilter" value="false"/>
    </translator>
```

Tip	Native queries The OData translator cannot perform native or direct query execution. However, you can use the *invokehttp* method of the Web services translator to issue REST-based calls, and then use SQLXML to parse results.

Tip	Using OData as server. Teiid can not only consume OData-based data sources, but it can also expose any data source as an OData-based web service.

For more information about configuring an OData server, see OData Support.

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 web services resource adapter for making web service calls. This translator is extension of *Web Services Translator*. The OData V4 translator is not for use with older OData V1-3 sources. Use the OData translator (*"odata"*) for older OData sources.

What is OData
The Open Data Protocol (OData) Web protocol is for querying and updating data that provides a way to unlock your data and free it from silos that exist in applications today. OData does this by applying and building upon Web technologies such as HTTP, Atom Publishing Protocol (AtomPub), and JSON to provide access to information from a variety of applications, services, and stores. OData is being used to expose and access information from a variety of sources including, but not limited to, relational databases, file systems, content management systems and traditional Web sites.

Using this specification from the OASIS group, with the help from the Olingo framework, 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 following table shows the mapping selections in the OData V4 translator from a CSDL document

Note	Using OData as a server Teiid can not only consume OData-based data sources, but it can expose any data source as an OData based web service. For more information see *OData Support* in the Client Developer’s Guide.

OData	Mapped to relational entity
EntitySet	Table
EntityType	Table see [1]
ComplexType	Table see [2]
FunctionImport	Procedure [3]
ActionImport	Procedure [3]
NavigationProperties	Table [4]

[1] Only if the EntityType is exposed as the EntitySet in the Entity container. [2] Only if the complex type is used as property in the exposed EntitySet. This table will be designed as child table with foreign key [1-to-1] or [1-to-many] relationship to the parent.
[3] If the return type is EntityType or ComplexType, the procedure is designed to return a table. [4] Navigation properties are exposed as tables. The table will be created with foreign key relationship to the parent.

All CRUD operations will be appropriately mapped to the resulting entity based on the SQL submitted to the OData translator.

Usage
Usage of a OData source is similar a JDBC translator. The metadata import is supported through the translator, once the metadata is imported from source system and exposed in relational terms, then this source can be queried as if the EntitySets, Function Imports and Action Imports were local to the Teiid system.

It is not recommended to define your own metadata using Teiid DDL for complex services. There are several extension metadata properties required to enable proper functioning. On non-string properties, a `NATIVE_TYPE` property is expected and should specify the full EDM type name - `Edm.xxx` .

The below is sample VDB that can read metadata service from TripPin service on http://odata.org site.

```
<vdb name="trippin" version="1">
    <model name="trippin">
         <source name="odata4" translator-name="odata4" connection-jndi-name="java:/tripDS"/>
    </model>
</vdb>
```

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>
```

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) ;
```

Execution properties

Sometimes default properties need to adjusted for proper execution of the translator. The following execution properties extend or limit the functionality of the translator based on the physical source capabilities.

Name	Description	Default
SupportsOdataCount	Supports $count	true
SupportsOdataFilter	Supports $filter	true
SupportsOdataOrderBy	Supports $orderby	true
SupportsOdataSkip	Supports $skip	true
SupportsOdataTop	Supports $top	true
SupportsUpdates	Supports INSERT/UPDATE/DELETE	true

The OData server that you connect to might not fully implement the entire OData specification. If the server’s OData implementation does not support a feature, set "execution properties" to turn off the corresponding capability, so that Teiid does not push down invalid queries to the translator.

```
<translator name="odata-override" type="odata">
     <property name="SupportsOdataFilter" value="false"/>
</translator>
```

then use "odata-override" as the translator name on your source model.

Importer properties
The following table lists the importer properties that define the behavior of the translator during metadata import from the physical source.

Name	Description	Default
schemaNamespace	Namespace of the schema to import	null

Example importer settings to only import tables and views from Trippin service exposed on odata.org

```
<property name="importer.schemaNamespace" value="Microsoft.OData.SampleService.Models.TripPin"/>
```

You can leave this property undefined. If the translator does not detect a configured instance of the property, it specifies the default name of the EntityContainer.

JCA resource adapter
The resource adapter for this translator is a Web Service Data Source.

Tip	**Native queries** - Native or direct query execution is not supported through the OData translator. However, you can use the *invokehttp* method of the Web services translator to issue REST-based calls, and then use SQLXML to parse results.

Swagger Translator

The Swagger translator, known by the type name *"swagger"* exposes Swagger data sources via realational concepts and uses the Teiid WS resource adapter for making web service calls.

Note	**What is 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. The swagger translator provides support for Swagger version 1/2 and OpenAPI version 2. See also the OpenAPI Translator.

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 a source system's swagger.json metadata file and exposed as stored procedures in Teiid. The source system can be queried by executing these stored procedures in Teiid system.

Note	Parameter order is guaranteed by the swagger libraries. However 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 properties 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

OpenAPI translator

The OpenAPI translator, known by the type name *"openapi"* exposes OpenAPI data sources via relational concepts and uses the Teiid WS resource adapter for making web service calls.

What is OpenAPI?
[OpenAPI is a simple yet powerful representation of your RESTful API. With the largest ecosystem of API tooling on the planet, thousands of developers are supporting OpenAPI in almost every modern programming language and deployment environment. With an OpenAPI-enabled API, you get interactive documentation, client SDK generation, and discoverability.

This translator is compatible with OpenAPI/Swagger v2 and OpenAPI v3.

Usage
Usage of a OpenAPI source is similar any other translator in Teiid. The translator enables metadata import. The metadata is imported from source system's metadata file and then exposed as stored procedures in Teiid. The source system can be queried by executing these stored procedures in Teiid system.

Note	Although parameter order is guaranteed by the Swagger libraries, if you rely upon the native import, it is best if you call procedures using named, rather than positional parameters.

The below is sample VDB that can read metadata from Petstore reference service on http://petstore.swagger.io/ site.

```
<vdb name="petstore" version="1">
    <model visible="true" name="m">
        <property name="importer.metadataUrl" value="/swagger.json"/>
        <source name="s" translator-name="openapi" connection-jndi-name="java:/openapi"/>
    </model>
</vdb>
```

The required resource-adapter configuration will look like

```
<resource-adapter id="openapi">
    <module slot="main" id="org.jboss.teiid.resource-adapter.webservice"/>
    <transaction-support>NoTransaction</transaction-support>
    <connection-definitions>
        <connection-definition class-name="org.teiid.resource.adapter.ws.WSManagedConnectionFactory" jndi-name=
"java:/openapi" enabled="true" use-java-context="true" pool-name="teiid-openapi-ds">
            <config-property name="EndPoint">
                http://petstore.swagger.io/v2
            </config-property>
        </connection-definition>
    </connection-definitions>
</resource-adapter>
```

After you configure the preceding resource-adapter and deploy the VDB successfully, then you can connect to the VDB deployed using Teiid JDBC driver and issue SQL statements such as the following:

```
EXEC findPetsByStatus(('sold',))
EXEC getPetById(1461159803)
EXEC deletePet('', 1461159803)
```

Execution properties
Execution properties extend/limit the functionality of the translator based on the physical source capabilities. Sometimes default properties must be adjusted for proper execution of the translator.

Execution properties

None.

Importer properties

The following table lists the importer properties that define the behavior of the translator during the import of from the physical source.

Name	Description	Default
metadataUrl	URL from which to obtain the OpenAPI metadata. May be a local file using a file: URL.	true
server	The server to use. Otherwise the first server listed will be used.	null
preferredProduces	Preferred Accept MIME type header, this should be one of the OpenAPI 'produces' types;	application/json
preferredConsumes	Preferred Content-Type MIME type header, this should be one of the OpenAPI 'consumer' types;	application/json

JCA resource adapter

The resource adapter for this translator is a Web Service Data Source.

Tip	**Native queries** - The OpenAPI translator cannot perform native or direct query execution. However, you can use the *invokehttp* method of the Web services translator to issue REST-based calls, and then use SQLXML to parse results.

Limitations

The OpenAPI translator does not fully implement all of the features of OpenAPI. The following limitations apply:

- You cannot set the MIME type to `application/xml` in either the `Accept` or `Content-Type` headers.
- File and Map properties cannot be used. As a result, any multi-part payloads are not supported.
- The translator does not process security metadata.
- The translator does not process custom properties that start with `x-` .
- The translator does not work with following JSON schema keywords:
    - `allOf`
    - `multipleOf`
    - `items`

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

You can use the Salesforce translator to run `SELECT` , `DELETE` , `INSERT` , `UPSERT,` and `UPDATE` operations against a Salesforce.com account.

salesforce
The translator, known by the type name ***salesforce***, 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 22.0 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.

Note	The default URL for a Salesforce source may change from release to release. Especially if you are relying on metadata import it is recommended that the Salesforce URL is configured on the source. The URL will contain an explicit API version which means the imported metadata will remain consistent.

Table 1. Execution properties

Name	Description	Default
MaxBulkInsertBatchSize	Batch Size to use to insert bulk inserts.	2048
SupportsGroupBy	Enables `GROUP BY` Pushdown. Set to false to have Teiid process group by aggregations, such as those returning more than 2000 rows which error in SOQL.	true

The Salesforce translator can import metadata.

Table 2. 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. This includes CreatedXXX, LastModifiedXXX, and SystemModstamp fields.	false	false

NOTE When both *includeTables* and *excludeTables* patterns are present during the import, the *includeTables* pattern matched first, then the *excludePatterns* will be applied.

Note	If you need connectivity to an API version other than what is built in, you may try to use an existing connectivity pair, but in some circumstances - especially accessing a later remote api from an older Java API - this is not possible and results in what appears to be hung connections.

Extension metadata properties

Salesforce is not relational database, however 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 in the following example:

```
CREATE FOREIGN TABLE Pricebook2 (
        Id string,
        Name string,
        IsActive boolean,
        IsStandard boolean,
        Description string,
        IsDeleted boolean)
        OPTIONS (
          UPDATABLE 'TRUE',
          "teiid_sf:Supports Query" 'TRUE');
```

In the preceding example, the property in the `OPTIONS` clause with the property `"teiid_sf:Supports Query"` set to `TRUE` indicates that you can run `SELECT` commands against this table. The following table lists the metadata extension properties that can be used in a Salesforce schema.

Property Name	Description	Required	Default	Applies To
Supports Query	You can run `SELECT` commands against the table.	false	true	Table
Supports Retrieve	You can retrieve the results of `SELECT` commands run against the table.	false	true	Table

SQL processing

Salesforce does not provide the same set of functionality as a relational database. For example, Salesforce does not support arbitrary joins between tables. However, working in combination with the Teiid Query Planner, the Salesforce connector can use nearly all of the SQL syntax capabilities in Teiid. The Salesforce Connector executes SQL commands by "pushing down" the command to Salesforce whenever possible, depending on the available capabilities. Teiid will automatically provide additional database functionality when the Salesforce Connector does not explicitly enable use of a given SQL construct. In cases where certain SQL capabilities cannot be pushed down to Salesforce, Teiid will push down the capabilities that it can, 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 issue queries with a `GROUP BY` clause, and you receive a Salesforce error that indicates that `queryMore` is not supported, you can either add limits, or set the execution property `SupportsGroupBy` to `false`.

```
SELECT array_agg(Reports) FROM Supervisor where Division = 'customer support';
```

Neither Salesforce, nor the Salesforce Connector support the `array_agg()` scalar. however, both are compatible with the `CompareCriteriaEquals` query, so the connector transforms the query that it receives into this query to Salesforce.

```
SELECT Reports FROM Supervisor where Division = 'customer support';
```

The array_agg() function will be applied by the Teiid Query Engine to the result set returned by the connector.

In some cases, multiple calls to the Salesforce application will be made to process the SQL that is passed to the connector.

```
DELETE From Case WHERE Status = 'Closed';
```

The API in Salesforce to delete objects can delete by object ID only. In order to accomplish this, the Salesforce connector will first execute a query to get the IDs of the correct objects, and then delete those objects. So the above DELETE command will result in the following two commands.

```
SELECT ID From Case WHERE Status = 'Closed';
DELETE From Case where ID IN (<result of query>);
```

NOTE The Salesforce API DELETE call is not expressed in SQL, but the above is an equivalent SQL expression.

It’s useful to be aware of incompatible capabilities, in order to avoid fetching large data sets from Salesforce and making you queries as performant as possible. For information about the SQL constructs that you can push down to Salesforce, see Compatible SQL capabilities.

Selecting from multi-select picklists
A multi-select picklist is a field type in Salesforce that can contain multiple values in a single field. Query criteria operators for fields of this type in SOQL are limited to EQ, NE, includes and excludes. For the Salesforce documentation about how to select from multi-select picklists, see Querying Multi-select Picklists

Teiid SQL does not support the includes or excludes operators, but the Salesforce connector provides user-defined function definitions for these operators that provide equivalent functionality for fields of type multi-select. The definition for the functions is:

```
boolean includes(Column column, String param)
boolean excludes(Column column, String param)
```

For example, take a single multi-select picklist column called Status that contains all of these values.

- current

- working
- critical

For that column, all of the below are valid queries:

```
SELECT * FROM Issue WHERE true = includes (Status, 'current, working' );
SELECT * FROM Issue WHERE true = excludes (Status, 'current, working' );
SELECT * FROM Issue WHERE true = includes (Status, 'current;working, critical' );
```

EQ and NE criteria will pass to Salesforce as supplied. For example, these queries will not be modified by the connector.

```
SELECT * FROM Issue WHERE Status = 'current';
SELECT * FROM Issue WHERE Status = 'current;critical';
SELECT * FROM Issue WHERE Status != 'current;working';
```

Selecting all objects
You can use the Salesforce connector to call the `queryAll` operation from the Salesforce API. The `queryAll` operation is equivalent to the query operation with the exception that it returns data about all current and deletedobjects in the system.

The connector determines if it will call the query or `queryAll` operation via reference to the `isDeleted` property present on each Salesforce object, and modeled as a column on each table generated by the importer. By default this value is set to `false` when the model is generated and thus the connector calls query. Users are free to change the value in the model to `true` , changing the default behavior of the connector to be `queryAll` .

The behavior is different if `isDeleted` is used as a parameter in the query. If the `isDeleted` column is used as a parameter in the query, and the value is `true` , then the connector calls `queryAll` .

```
select * from Contact where isDeleted = true;
```

If the `isDeleted` column is used as a parameter in the query, and the value is `false` , then the connector that performs the default behavior will call the query.

```
select * from Contact where isDeleted = false;
```

Selecting updated objects
If the option is selected when importing metadata from Salesforce, a GetUpdated procedure is generated in the model with the following structure:

```
GetUpdated (ObjectName IN string,
    StartDate IN datetime,
    EndDate IN datetime,
    LatestDateCovered OUT datetime)
returns
    ID string
```

See the description of the GetUpdated operation in the Salesforce documentation for usage details.

Selecting deleted objects
If the option is selected when importing metadata from Salesforce, a GetDeleted procedure is generated in the model with the following structure:

```
GetDeleted (ObjectName IN string,
    StartDate IN datetime,
    EndDate IN datetime,
    EarliestDateAvailable OUT datetime,
```

```
    LatestDateCovered OUT datetime)
returns
    ID string,
    DeletedDate datetime
```

See the description of the GetDeleted operation in the Salesforce documentation for usage details.

Relationship queries

Unlike a relational database, Salesforce does not support join operations, but it does have support for queries that include parent-to-child or child-to-parent relationships between objects. These are termed Relationship Queries. You can run Relationship Queries in the SalesForce connector through Outer Join syntax.

```
SELECT Account.name, Contact.Name from Contact LEFT OUTER JOIN Account
on Contact.Accountid = Account.id
```

This query shows the correct syntax to query a SalesForce model with to produce a relationship query from child to parent. It resolves to the following query to SalesForce.

```
SELECT Contact.Account.Name, Contact.Name FROM Contact
```

```
select Contact.Name, Account.Name from Account Left outer Join Contact
on Contact.Accountid = Account.id
```

This query shows the correct syntax to query a SalesForce model with to produce a relationship query from parent to child. It resolves to the following query to SalesForce.

```
SELECT Account.Name, (SELECT Contact.Name FROM
Account.Contacts) FROM Account
```

See the description of the Relationship Queries operation in the SalesForce documentation for limitations.

Bulk insert queries

You can also use bulk insert statements in the SalesForce translator by using JDBC batch semantics or SELECT INTO semantics. The batch size is determined by the execution property *MaxBulkInsertBatchSize*, which can be overridden in the vdb file. The default value of the batch is 2048. The bulk insert feature uses the async REST based API exposed by Salesforce for execution for better performance.

Bulk selects

When querying tables with more than 10,000,000 records, or if experiencing timeouts with just result batching, Teiid can issue queries to Salesforce using the bulk API. When using a bulk select, primary key (PK) chunking is enabled if it is compatible with the query.

The use of the bulk api requires a source hint in the query:

```
SELECT /*+ sh salesforce:'bulk' */ Name ... FROM Account
```

Where salesforce is the source name of the target source.

The default chunk size of 100,000 records will be used.

Note	This feature is only supported in the Salesforce API version 28 or higher.

Compatible SQL capabilities

You ca use the following SQL capabilities with the Salesforce Connector. These SQL constructs will be pushed down to Salesforce.

- SELECT command
- INSERT Command
- UPDATE Command
- DELETE Command
- NotCriteria
- OrCriteria
- CompareCriteriaEquals
- CompareCriteriaOrdered
- IsNullCritiera
- InCriteria
- LikeCriteria - Can be used for String fields only.
- RowLimit
- Basic Aggregates
- OuterJoins with join criteria KEY

Native Queries
Salesforce procedures may optionally have native queries associated with them. For more information, see *Parameterizable native queries* in Translators. The operation prefix (select;, insert;, update;, delete; - see below for more) must be present in the native-query, but it will not be issued as part of the query to the source.

Example DDL for a Salesforce native procedure

```
CREATE FOREIGN PROCEDURE proc (arg1 integer, arg2 string) OPTIONS ("teiid_rel:native-query" 'search;SELECT ...
complex SOQL ... WHERE col1 = $1 and col2 = $2')
returns (col1 string, col2 string, col3 timestamp);
```

Direct query procedure
This feature is turned off by default because of the security risk this exposes to execute any command against the source. To enable direct query procedures, set the execution property called `SupportsDirectQueryProcedure` to `true`. For more information, see *Override the execution properties* in [as_translators].

Tip	By default the name of the procedure that executes the queries directly is called **native**. For information about how to change the default name, see *Override the execution properties* in [as_translators].

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 example

```
SELECT x.* FROM (call sf_source.native('search;SELECT Account.Id, Account.Type, Account.Name FROM Account')) w,
 ARRAYTABLE(w.tuple COLUMNS "id" string , "type" string, "name" String) AS x
```

from the above code, the "search" keyword followed by a query statement.

Note	The SOQL is treated as a parameterized native query so that parameter values may be inserted in the query string properly. For more information, see *Parameterizable native queries* in Translators. The results returned by search may contain the object Id as the first column value regardless of whether it was selected. Also queries that select columns from multiple object types will not be correct.

Delete Example

```
SELECT x.* FROM (call sf_source.native('delete;', 'id1', 'id2')) w,
 ARRAYTABLE(w.tuple COLUMNS "updatecount" integer) AS x
```

form the above code, the "delete;" keyword followed by the ids to delete as varargs.

Create example

```
SELECT x.* FROM
 (call sf_source.native('create;type=table;attributes=one,two,three', 'one', 2, 3.0)) w,
 ARRAYTABLE(w.tuple COLUMNS "update_count" integer) AS x
```

form the above code, the "create" or "update" keyword must be followed by the following properties. Attributes must be matched positionally by the procedure variables - thus in the example attribute two will be set to 2.

Property Name	Description	Required
type	Table Name	Yes
attributes	comma separated list of names of the columns	no

The values for each attribute is specified as separate argument to the "native" procedure.

Update is similar to create, with one more extra property called "id", which defines identifier for the record.

Update example

```
SELECT x.* FROM
 (call sf_source.native('update;id=pk;type=table;attributes=one,two,three', 'one', 2, 3.0)) w,
 ARRAYTABLE(w.tuple COLUMNS "update_count" integer) AS x
```

Tip	By default the name of the procedure that executes the queries directly is called native, however you can add set an override execution property in the DDL file to change it.

JCA resource adapter

The resource adapter for this translator is provided through Salesforce data sources. For configuration information, see *Salesforce Data Sources* in the Administrator’s Guide.

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/SOAP services. Results from this translator will commonly be used with the TEXTTABLE or XMLTABLE table functions to use CSV or XML formated data.

Execution properties

Name	Description	When Used	Default
DefaultBinding	The binding that should be used if one is not specified. Can be one of HTTP, SOAP11, or SOAP12.	invoke*	SOAP12
DefaultServiceMode	The default service mode. For SOAP, MESSAGE mode indicates that the request will contain the entire SOAP envelope. and not just the contents of the SOAP body. Can be one of MESSAGE or PAYLOAD	invoke* or WSDL call	PAYLOAD
XMLParamName	Used with the HTTP binding (typically with the GET method) to indicate that the request document should be part of the query string.	invoke*	null - unused

Note	Setting the proper binding value on the translator is recommended as it removes the need for callers to pass an explicit value. If your service is actually uses SOAP11, but the binding used SOAP12 you will receive execution failures.

There are no importer settings, but it can provide metadata for VDBs. If the connection is configured to point at a specific WSDL, the translator will import all SOAP operations under the specified service and port as procedures.

Importer properties

When specifying the importer property, it must be prefixed with "importer.". Example: importer.tableTypes

Name	Description	Default
importWSDL	Import the metadata from the WSDL URL configured in resource-adapter.	true

Usage

The translator exposes low level procedures for accessing web services. 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 source configuration. The endpoint URL may be absolute or relative. If it's relative then it will be combined with the default endpoint.

Since multiple parameters are not required to have values, it is often more clear to call the invoke procedure with named parameter syntax.

```
call invoke(binding=>'HTTP', action=>'GET')
```

The request XML should be a valid XML document or root element.

InvokeHTTP procedure
`invokeHttp` can return the byte contents of an HTTP(S) call.

```
Procedure invokeHttp(action in STRING, request in OBJECT, endpoint in STRING, stream in BOOLEAN, contentType 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 source configuration. The endpoint URL may be absolute or relative. If it's relative then it will be combined with the default endpoint.

Since multiple parameters are not required to have values, it is often more clear to call the invokeHttp procedure with named parameter syntax.

```
call invokeHttp(action=>'GET')
```

The request can be one of SQLXML, STRING, BLOB, or CLOB. The request will be sent as the POST payload in byte form. For STRING/CLOB values this will default to the UTF-8 encoding. To control the byte encoding, see the to_bytes function.

The optional headers parameter can be used to specify the request header values as a JSON value. The JSON value should be a JSON object with primitive or list of primitive values.

```
call invokeHttp(... headers=>jsonObject('application/json' as "Content-Type", jsonArray('gzip', 'deflate') as "
Accept-Encoding"))
```

Recommendations for setting headers parameter:

- `Content-Type` might be necessary if the HTTP POST/PUT method is invoked.
- *Accept* is necessary if you want to control return Media Type.

WSDL based procedures
The procedures above give you anonymous way to execute any web service methods by supplying an endpoint, with this mechanism you can alter the endpoint defined in WSDL with a different endpoint. However, if you have access to the WSDL, then you can configure the WSDL URL in the web-service resource-adapter's connection configuration, Web Service translator can parse the WSDL and provide the methods under configured port as pre-built procedures as its metadata. If you are using the default native metadata import, you will see the procedures in your web service's source model.

Note	Native queries You cannot use native queries or direct query execution procedures with the web services translator.

Streaming considerations

If the stream parameter is set to `true`, then the resulting LOB value may only be used a single time. If stream is `null` or `false`, then the engine may need to save a copy of the result for repeated use. Care must be used as some operations, such as casting or `XMLPARSE` might perform validation which results in the stream being consumed.

JCA resource adapter
The resource adapter for this translator is a Web Service Data Source.

The WS resource adapter may optionally be configured to point at a specific WSDL.

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

At its core, Teiid is a federated, relational query engine. This query engine allows you to treat all of your data sources as one virtual database, and access them through a single SQL query. As a result, instead of focusing on hand-coding joins, you can focus on building your application, and on running other relational operations between data sources.

Planning Overview

When the query engine receives an incoming SQL query it performs the following operations:

1. **Parsing** — Validates syntax and convert to internal form.
2. **Resolving** — Links all identifiers to metadata and functions to the function library.
3. **Validating** — Validates SQL semantics based on metadata references and type signatures.
4. **Rewriting** — Rewrites SQL to simplify expressions and criteria.
5. **Logical plan optimization** — Converts the rewritten canonical SQL to a logical plan for in-depth optimization. The 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 the `SET SHOWPLAN DEBUG`clause, as described in the Client Development Guide. For sample steps, see *Reading a debug plan* in Query Planner. For more information about logical plan nodes and rule-based optimization, see Query Planner.
6. **Processing plan conversion** — Converts the logic plan to an executable form where the nodes represent basic processing operations. The final processing plan is displayed as a query plan. For more information, see Query plans.

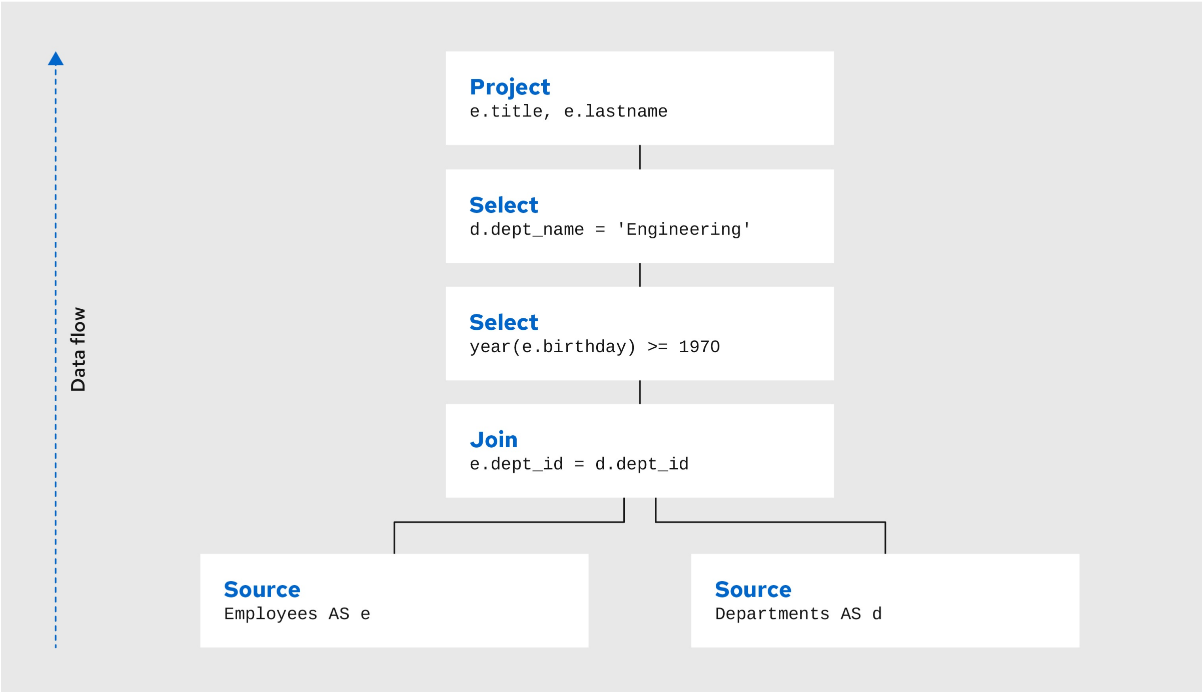
The logical query plan is a tree of operations that is used to transform data in source tables to the expected result set. In the tree, data flows from the bottom (tables) to the top (output). The primary logical operations are *select* (select or filter rows based on a criteria), *project* (project or compute column values), *join*, *source* (retrieve data from a table), *sort* (ORDER BY), *duplicate removal* (SELECT DISTINCT), *group* (GROUP BY), and *union* (UNION).

For example, consider the following query that retrieves all engineering employees born since 1970.

Example query

```
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* — it is not how the plan is actually executed. Starting from this initial plan, the query planner performs transformations on the query plan tree to produce an equivalent plan that retrieves the same results faster. Both a federated query planner and a relational database planner deal with the same concepts and many of the same plan transformations. In this example, the criteria on the Departments and Employees tables will be pushed down the tree to filter the results as early as possible.

In both cases, the goal is to retrieve the query results in the fastest possible time. However, the relational database planner achieve this primarily by optimizing the access paths in pulling data from storage.

In contrast, a federated query planner is less concerned about storage access, because it is typically pushing that burden to the data source. The most important consideration for a federated query planner is minimizing data transfer.

Query planner

For each sub-command in the user command an appropriate kind of sub-planner is used (relational, XML, procedure, etc).

Each planner has three primary phases:

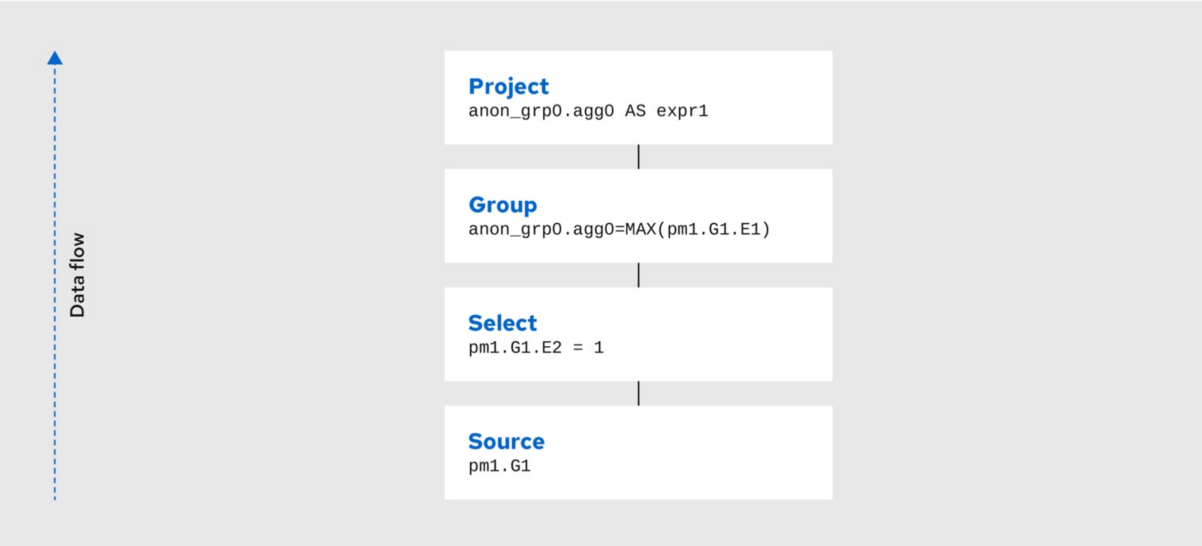
1. Generate canonical plan
2. Optimization
3. Plan to process converter — Converts plan data structure into a processing form.

Relational planner
A relational processing plan is created by the optimizer after the logical plan is manipulated by a series of rules. The application of rules is determined both by the query structure and by the rules themselves. The node structure of the debug plan resembles that of the processing plan, but the node types more logically represent SQL operations.

Canonical plan and all nodes
As described in the Planning overview, a SQL statement submitted to the query engine is parsed, resolved, validated, and rewritten before it is converted into a canonical plan form. The canonical plan form most closely resembles the initial SQL structure. A SQL select query has the following possible clauses (all but SELECT are optional): WITH, SELECT, FROM, WHERE, GROUP BY, HAVING, ORDER BY, LIMIT. These clauses are logically executed in the following order:

1. WITH (create common table expressions) — Processed by a specialized PROJECT NODE.
2. FROM (read and join all data from tables) — Processed by a SOURCE node for each from clause item, or a Join node (if >1 table).
3. WHERE (filter rows) — Processed by a SELECT node.
4. GROUP BY (group rows into collapsed rows) — Processed by a GROUP node.
5. HAVING (filter grouped rows) — Processed by a SELECT node.
6. SELECT (evaluate expressions and return only requested rows) — Processed by a PROJECT node and DUP_REMOVE node (for SELECT DISTINCT).
7. INTO — Processed by a specialized PROJECT with a SOURCE child.
8. ORDER BY (sort rows) — Processed by a SORT node.
9. LIMIT (limit result set to a certain range of results) — Processed by a LIMIT node.

For example, a SQL statement such as `SELECT max(pm1.g1.e1) FROM pm1.g1 WHERE e2 = 1` creates a logical plan:




```
Project(groups=[anon_grp0], props={PROJECT_COLS=[anon_grp0.agg0 AS expr1]})
  Group(groups=[anon_grp0], props={SYMBOL_MAP={anon_grp0.agg0=MAX(pm1.G1.E1)}})
    Select(groups=[pm1.G1], props={SELECT_CRITERIA=pm1.G1.E2 = 1})
      Source(groups=[pm1.G1])
```

Here the Source corresponds to the FROM clause, the Select corresponds to the WHERE clause, the Group corresponds to the implied grouping to create the max aggregate, and the Project corresponds to the SELECT clause.

Note	The effect of grouping generates what is effectively an inline view, `anon_grp0`, to handle the projection of values created by the grouping.

Table 1. **Node Types**

Type Name	Description
ACCESS	A source access or plan execution.
DUP_REMOVE	Removes duplicate rows
JOIN	A join (LEFT OUTER, FULL OUTER, INNER, CROSS, SEMI, and so forth).
PROJECT	A projection of tuple values
SELECT	A filtering of tuples
SORT	An ordering operation, which may be inserted to process other operations such as joins.
SOURCE	Any logical source of tuples including an inline view, a source access, XMLTABLE, and so forth.
GROUP	A grouping operation.
SET_OP	A set operation (UNION/INTERSECT/EXCEPT).
NULL	A source of no tuples.

TUPLE_LIMIT	Row offset / limit

Node properties

Each node has a set of applicable properties that are typically shown on the node.

Table 2. **Access properties**

Property Name	Description
ATOMIC_REQUEST	The final form of a source request.
MODEL_ID	The metadata object for the target model/schema.
PROCEDURE_CRITERIA/PROCEDURE_INPUTS/PROCEDURE_DEFAULTS	Used in planning procedureal relational queries.
IS_MULTI_SOURCE	set to true when the node represents a multi-source access.
SOURCE_NAME	used to track the multi-source source name.
CONFORMED_SOURCES	tracks the set of conformed sources when the conformed extension metadata is used.
SUB_PLAN/SUB_PLANS	used in multi-source planning.

Table 3. **Set operation properties**

Property Name	Description
SET_OPERATION/USE_ALL	defines the set operation(UNION/INTERSECT/EXCEPT) and if all rows or distinct rows are used.

Table 4. **Join properties**

Property Name	Description
JOIN_CRITERIA	All join predicates.
JOIN_TYPE	Type of join (INNER, LEFT OUTER, and so forth).
JOIN_STRATEGY	The algorithm to use (nested loop, merge, and so forth).
LEFT_EXPRESSIONS	The expressions in equi-join predicates that originate from the left side of the join.
RIGHT_EXPRESSIONS	The expressions in equi-join predicates that originate from the right side of the join.
DEPENDENT_VALUE_SOURCE	set if a dependent join is used.
NON_EQUI_JOIN_CRITERIA	Non-equi join predicates.
SORT_LEFT	If the left side needs sorted for join processing.

SORT_RIGHT	If the right side needs sorted for join processing.
IS_OPTIONAL	If the join is optional.
IS_LEFT_DISTINCT	If the left side is distinct with respect to the equi join predicates.
IS_RIGHT_DISTINCT	If the right side is distinct with respect to the equi join predicates.
IS_SEMI_DEP	If the dependent join represents a semi-join.
PRESERVE	If the preserve hint is preserving the join order.

Table 5. **Project properties**

Property Name	Description
PROJECT_COLS	The expressions projected.
INTO_GROUP	The group targeted if this is a select into or insert with a query expression.
HAS_WINDOW_FUNCTIONS	True if window functions are used.
CONSTRAINT	The constraint that must be met if the values are being projected into a group.
UPSERT	If the insert is an upsert.

Table 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.

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.

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, and so forth.) 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.

Table 9. **Group properties**

Property Name	Description
GROUP_COLS	The grouping columns.
ROLLUP	If the grouping includes a rollup.

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.

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, and so forth.
EST_SELECTIVITY	Represents the selectivity of a criteria node.

Rules

Relational optimization is based upon rule execution that evolves the initial plan into the execution plan. There are a set of pre-defined rules that are dynamically assembled into a rule stack for every query. The rule stack is assembled based on the contents of the user’s query and the views/procedures accessed. For example, if there are no view layers, then rule Merge Virtual, which merges view layers together, is not needed and will not be added to the stack. This allows the rule stack to reflect the complexity of the query.

Logically the plan node data structure represents a tree of nodes where the source data comes up from the leaf nodes (typically Access nodes in the final plan), flows up through the tree and produces the user’s results out the top. The nodes in the plan structure can have bidirectional links, dynamic properties, and allow any number of child nodes. Processing plans in contrast typically have fixed properties.

Plan rule manipulate the plan tree, fire other rules, and drive the optimization process. Each rule is designed to perform a narrow set of tasks. Some rules can be run multiple times. Some rules require a specific set of precursors to run properly.

- Access Pattern Validation — Ensures that all access patterns have been satisfied.

- Apply Security — Applies row and column level security.
- Assign Output Symbol — This rule walks top down through every node and calculates the output columns for each node. Columns that are not needed are dropped at every node, which is known as projection minimization. This is done by keeping track of both the columns needed to feed the parent node and also keeping track of columns that are "created" at a certain node.
- Calculate Cost — Adds costing information to the plan
- Choose Dependent — This rule looks at each join node and determines whether the join should be made dependent and in which direction. Cardinality, the number of distinct values, and primary key information are used in several formulas to determine whether a dependent join is likely to be worthwhile. The dependent join differs in performance ideally because a fewer number of values will be returned from the dependent side.

  Also, we must consider the number of values passed from independent to dependent side. If that set is larger than the maximum number of values in an IN criteria on the dependent side, then we must break the query into a set of queries and combine their results. Executing each query in the connector has some overhead and that is taken into account. Without costing information a lot of common cases where the only criteria specified is on a non-unique (but strongly limiting) field are missed.

  A join is eligible to be dependent if:

  - There is at least one equi-join criterion, for example, `tablea.col = tableb.col`
  - The join is not a full outer join and the dependent side of the join is on the inner side of the join.

The join will be made dependent if one of the following conditions, listed in precedence order, holds:

- There is an unsatisfied access pattern that can be satisfied with the dependent join criteria.
- The potential dependent side of the join is marked with an option makedep.
- (4.3.2) if costing was enabled, the estimated cost for the dependent join (5.0+ possibly in each direction in the case of inner joins) is computed and compared to not performing the dependent join. If the costs were all determined (which requires all relevant table cardinality, column ndv, and possibly nnv values to be populated) the lowest is chosen.
- If key metadata information indicates that the potential dependent side is not "small" and the other side is "not small" or (5.0.1) the potential dependent side is the inner side of a left outer join.

Dependent join is the key optimization we use to efficiently process multi-source joins. Instead of reading all of source A and all of source B and joining them on A.x = B.x, we read all of A, and then build a set of A.x that are passed as a criteria when querying B. In cases where A is small and B is large, this can drastically reduce the data retrieved from B, thus greatly speeding the overall query.

- Choose Join Strategy — Choose the join strategy based upon the cost and attributes of the join.
- Clean Criteria — Removes phantom criteria.
- Collapse Source — Takes all of the nodes below an access node and creates a SQL query representation.
- Copy Criteria — This rule copies criteria over an equality criteria that is present in the criteria of a join. Since the equality defines an equivalence, this is a valid way to create a new criteria that may limit results on the other side of the join (especially in the case of a multi-source join).
- Decompose Join — This rule performs a partition-wise join optimization on joins of a partitioned union. For more information, see *Partitioned unions* in Federated optimizations. The decision to decompose is based upon detecting that each side of the join is a partitioned union (note that non-ANSI joins of more than 2 tables may cause the optimization to not detect the appropriate join). The rule currently only looks for situations where at most 1 partition matches from each side.
- Implement Join Strategy — Adds necessary sort and other nodes to process the chosen join strategy

- Merge Criteria — Combines select nodes
- Merge Virtual — Removes view and inline view layers
- Place Access — Places access nodes under source nodes. An access node represents the point at which everything below the access node gets pushed to the source or is a plan invocation. Later rules focus on either pushing under the access or pulling the access node up the tree to move more work down to the sources. This rule is also responsible for placing access patterns. For more information, see *Access patterns* in Federated optimizations
- Plan Joins — This rule attempts to find an optimal ordering of the joins performed in the plan, while ensuring that access pattern dependencies are met. This rule has three main steps.
  1. It must determine an ordering of joins that satisfy the access patterns present.
  2. It will heuristically create joins that can be pushed to the source (if a set of joins are pushed to the source, we will not attempt to create an optimal ordering within that set. More than likely it will be sent to the source in the non-ANSI multi-join syntax and will be optimized by the database).
  3. It will use costing information to determine the best left-linear ordering of joins performed in the processing engine. This third step will do an exhaustive search for 7 or less join sources and is heuristically driven by join selectivity for 8 or more sources.
- Plan Outer Joins — Reorders outer joins as permitted to improve push down.
- Plan Procedures — Plans procedures that appear in procedural relational queries.
- Plan Sorts — Optimizations around sorting, such as combining sort operations or moving projection.
- Plan Subqueries — New for 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 the translator can process directly to be processed as a dependent set.

One of the most important optimization related to pushing criteria, is how the criteria will be pushed through join. Consider the following plan tree that represents a subtree of the plan for the query `select * from A inner join b on (A.x = B.x) where B.y = 3`

```
SELECT (B.y = 3)
        |
       JOIN - Inner Join on (A.x = B.x)
      /     \
   SRC (A)   SRC (B)
```

Note	SELECT nodes represent criteria, and SRC stands for SOURCE.

It is always valid for inner join and cross joins to push (single source) criteria that are above the join, below the join. This allows for criteria originating in the user query to eventually be present in source queries below the joins. This result can be represented visually as:

```
    JOIN - Inner Join on (A.x = B.x)
          /    \
         /   SELECT (B.y = 3)
        |         |
      SRC (A)   SRC (B)
```

The same optimization is valid for criteria specified against the outer side of an outer join. For example:

```
     SELECT (B.y = 3)
           |
          JOIN - Right Outer Join on (A.x = B.x)
         /     \
      SRC (A)   SRC (B)
```

Becomes

```
          JOIN - Right Outer Join on (A.x = B.x)
          /    \
         /   SELECT (B.y = 3)
        |         |
      SRC (A)   SRC (B)
```

However criteria specified against the inner side of an outer join needs special consideration. The above scenario with a left or full outer join is not the same. For example:

```
      SELECT (B.y = 3)
           |
          JOIN - Left Outer Join on (A.x = B.x)
         /     \
      SRC (A)   SRC (B)
```

Can become (available only after 5.0.2):

```
    JOIN - Inner Join on (A.x = B.x)
          /    \
         /   SELECT (B.y = 3)
        |         |
      SRC (A)   SRC (B)
```

Since the criterion is not dependent upon the null values that may be populated from the inner side of the join, the criterion is eligible to be pushed below the join — but only if the join type is also changed to an inner join. On the other hand, criteria that are dependent upon the presence of null values CANNOT be moved. For example:

```
    SELECT (B.y is null)
           |
          JOIN - Left Outer Join on (A.x = B.x)
         /     \
      SRC (A)   SRC (B)
```

The preceding plan tree must have the criteria remain above the join, becuase the outer join may be introducing null values itself.

- Raise Access — This rule attempts to raise the Access nodes as far up the plan as posssible. This is mostly done by looking at the source’s capabilities and determining whether the operations can be achieved in the source or not.

- Raise Null — Raises null nodes. Raising a null node removes the need to consider any part of the old plan that was below the null node.
- Remove Optional Joins — Removes joins that are marked as or determined to be optional.
- Substitute Expressions — Used only when a function based index is present.
- Validate Where All — Ensures criteria is used when required by the source.

Cost calculations
The cost of node operations is primarily determined by an estimate of the number of rows (also referred to as cardinality) that will be processed by it. The optimizer will typically compute cardinalities from the bottom up of the plan (or subplan) at several points in time with planning — once generally with rule calculate cost, and then specifically for join planning and other decisions. The cost calculation is mainly directed by the statistics set on physical tables (cardinality, NNV, NDV, and so forth) and is also influenced by the presence of constraints (unique, primary key, index, and so forth). If there is a situation that seems like a sub-optimal plan is being chosen, you should first ensure that at least representative table cardinalities are set on the physical tables involved.

Reading a debug plan
As each relational sub plan is optimized, the plan will show what is being optimized and it’s canonical form:

```
OPTIMIZE:
SELECT e1 FROM (SELECT e1 FROM pm1.g1) AS x

----------------------------------------------------------------------------
GENERATE CANONICAL:
SELECT e1 FROM (SELECT e1 FROM pm1.g1) AS x

CANONICAL PLAN:
Project(groups=[x], props={PROJECT_COLS=[e1]})
  Source(groups=[x], props={NESTED_COMMAND=SELECT e1 FROM pm1.g1, SYMBOL_MAP={x.e1=e1}})
    Project(groups=[pm1.g1], props={PROJECT_COLS=[e1]})
      Source(groups=[pm1.g1])
```

With more complicated user queries, such as a procedure invocation or one containing subqueries, the sub-plans may be nested within the overall plan. Each plan ends by showing the final processing plan:

```
----------------------------------------------------------------------------
OPTIMIZATION COMPLETE:
PROCESSOR PLAN:
AccessNode(0) output=[e1] SELECT g_0.e1 FROM pm1.g1 AS g_0
```

The affect of rules can be seen by the state of the plan tree before and after the rule fires. For example, the debug log below shows the application of rule merge virtual, which will remove the "x" inline view layer:

```
EXECUTING AssignOutputElements

AFTER:
Project(groups=[x], props={PROJECT_COLS=[e1], OUTPUT_COLS=[e1]})
  Source(groups=[x], props={NESTED_COMMAND=SELECT e1 FROM pm1.g1, SYMBOL_MAP={x.e1=e1}, OUTPUT_COLS=[e1]})
    Project(groups=[pm1.g1], props={PROJECT_COLS=[e1], OUTPUT_COLS=[e1]})
      Access(groups=[pm1.g1], props={SOURCE_HINT=null, MODEL_ID=Schema name=pm1, nameInSource=null, uuid=3335,
OUTPUT_COLS=[e1]})
        Source(groups=[pm1.g1], props={OUTPUT_COLS=[e1]})


============================================================================
EXECUTING MergeVirtual

AFTER:
Project(groups=[pm1.g1], props={PROJECT_COLS=[e1], OUTPUT_COLS=[e1]})
```

```
  Access(groups=[pm1.g1], props={SOURCE_HINT=null, MODEL_ID=Schema name=pm1, nameInSource=null, uuid=3335, 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 following code snippet shows that the access node could not be raised, because the parent `SELECT` node contained an unsupported subquery.

```
Project(groups=[pm1.g1], props={PROJECT_COLS=[e1], OUTPUT_COLS=null})
  Select(groups=[pm1.g1], props={SELECT_CRITERIA=e1 IN /*+ NO_UNNEST */ (SELECT e1 FROM pm2.g1), OUTPUT_COLS=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. For more information, see XQuery optimization. Document projection is the most common optimization. It will be shown in the debug plan as an annotation. For example, with the user query that contains "xmltable('/a/b' passing doc columns x string path '@x', val string path '.')", the debug plan would show a tree of the document that will effectively be used by the context and path XQuerys:

```
MEDIUM XQuery Planning Projection conditions met for /a/b - Document projection will be used
child element(Q{}a)
  child element(Q{}b)
    attribute attribute(Q{}x)
      child text()
    child text()
```

Query plans

When integrating information using a federated query planner it is useful to view the query plans to better understand how information is being accessed and processed, and to troubleshoot problems.

A query plan (also known as an execution or processing plan) is a set of instructions created by a query engine for executing a command submitted by a user or application. The purpose of the query plan is to execute the user's query in as efficient a way as possible.

Getting a query plan
You can get a query plan any time you execute a command. The following SQL options are available:

SET SHOWPLAN [ON|DEBUG]- Returns the processing plan or the plan and the full planner debug log. For more information, see *Reading a debug plan* in Query planner and *SET statement* in the Client Developer's Guide. With the above options, the query plan is available from the Statement object by casting to the `org.teiid.jdbc.TeiidStatement` interface or by using the SHOW PLAN statement. For more information, see *SHOW Statement* in the Client Developer's Guide. Alternatively you may use the EXPLAIN statement. For more information, see, Explain statement.

Retrieving a query plan using 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"));
```

Retrieving a query plan using explain

```
ResultSet rs = statement.executeQuery("explain select ...");
System.out.println(rs.getString("QUERY PLAN"));
```

The query plan is made available automatically in several of Teiid's tools.

Analyzing a query plan
After you obtain a query plan, you can examine it for the following items:

- Source pushdown — What parts of the query that got pushed to each source
  - Ensure that any predicates especially against indexes are pushed
- Joins — As federated joins can be quite expensive
  - Join ordering — Typically influenced by costing
  - Join criteria type mismatches.
  - Join algorithm used — Merge, enhanced merge, nested loop, and so forth.
- Presence of federated optimizations, such as dependent joins.

- Ensure hints have the desired affects. For more information about using hints, see the following additional resources:
    - *Hints and Options* in the Caching Guide.
    - *FROM clause hints* in FROM clause.
    - Subquery optimization.
    - Federated optimizations.

You can determine all of information in the preceding list from the processing plan. You will typically be interested in analyzing the textual form of the final processing plan. To understand why particular decisions are made for debugging or support you will want to obtain the full debug log which will contain the intermediate planning steps as well as annotations as to why specific pushdown decisions are made.

A query plan consists of a set of nodes organized in a tree structure. If you are executing a procedure, the overall query plan will contain additional information related the surrounding procedural execution.

In a procedural context the ordering of child nodes implies the order of execution. In most other situation, child nodes may be executed in any order even in parallel. Only in specific optimizations, such as dependent join, will the children of a join execute serially.

Relational query plans
Relational plans represent the processing plan that is composed of nodes representing building blocks of logical relational operations. Relational processing plans differ from logical debug relational plans in that they will contain additional operations and execution specifics that were chosen by the optimizer.

The nodes for a relational query plan are:

- Access — Access a source. A source query is sent to the connection factory associated with the source. (For a dependent join, this node is called Dependent Access.)
- Dependent procedure access — Access a stored procedure on a source using multiple sets of input values.
- Batched update — Processes a set of updates as a batch.
- Project — Defines the columns returned from the node. This does not alter the number of records returned.
- Project into — Like a normal project, but outputs rows into a target table.
- Insert plan execution — Similar to a project into, but executes a plan rather than a source query. Typically created when executing an insert into view with a query expression.
- Window function project — Like a normal project, but includes window functions.
- Select — Select is a criteria evaluation filter node (WHERE / HAVING).
- Join — Defines the join type, join criteria, and join strategy (merge or nested loop).
- Union all — There are no properties for this node, it just passes rows through from it’s children. Depending upon other factors, such as if there is a transaction or the source query concurrency allowed, not all of the union children will execute in parallel.
- Sort — Defines the columns to sort on, the sort direction for each column, and whether to remove duplicates or not.
- Dup remove — Removes duplicate rows. The processing uses a tree structure to detect duplicates so that results will effectively stream at the cost of IO operations.
- Grouping — Groups sets of rows into groups and evaluates aggregate functions.
- Null — A node that produces no rows. Usually replaces a Select node where the criteria is always false (and whatever tree is underneath). There are no properties for this node.

- Plan execution — Executes another sub plan. Typically the sub plan will be a non-relational plan.
- Dependent procedure execution — Executes a sub plan using multiple sets of input values.
- Limit — Returns a specified number of rows, then stops processing. Also processes an offset if present.
- XML table — Evaluates XMLTABLE. The debug plan will contain more information about the XQuery/XPath with regards to their optimization. For more information, see XQuery optimization.
- Text table - Evaluates TEXTTABLE
- Array table - Evaluates ARRAYTABLE
- Object table - Evaluates OBJECTTABLE

Node statistics
Every node has a set of statistics that are output. These can be used to determine the amount of data flowing through the node. Before execution a processor plan will not contain node statistics. Also the statistics are updated as the plan is processed, so typically you'll want the final statistics after all rows have been processed by the client.

Statistic	Description	Units
Node output rows	Number of records output from the node.	count
Node next batch process time	Time processing in this node only.	millisec
Node cumulative next batch process time	Time processing in this node + child nodes.	millisec
Node cumulative process time	Elapsed time from beginning of processing to end.	millisec
Node next batch calls	Number of times a node was called for processing.	count
Node blocks	Number of times a blocked exception was thrown by this node or a child.	count

In addition to node statistics, some nodes display cost estimates computed at the node.

Cost Estimates	Description	Units
Estimated node cardinality	Estimated number of records that will be output from the node; -1 if unknown.	count

The root node will display additional information.

Top level statistics	Description	Units
Data Bytes Sent	The size of the serialized data result (row and lob values) sent to the client.	bytes

Reading a processor plan

The query processor plan can be obtained in a plain text or XML format. The plan text format is typically easier to read, while the XML format is easier to process by tooling. When possible tooling should be used to examine the plans as the tree structures can be deeply nested.

Data flows from the leafs of the tree to the root. Sub plans for procedure execution can be shown inline, and are differentiated by different indentation. Given a user query of `SELECT pm1.g1.e1, pm1.g2.e2, pm1.g3.e3 from pm1.g1 inner join (pm1.g2 left outer join pm1.g3 on pm1.g2.e1=pm1.g3.e1) on pm1.g1.e1=pm1.g3.e1`, the text for a processor plan that does not push down the joins would look like:

```
ProjectNode
  + Output Columns:
    0: e1 (string)
    1: e2 (integer)
    2: e3 (boolean)
  + Cost Estimates:Estimated Node Cardinality: -1.0
  + Child 0:
    JoinNode
      + Output Columns:
        0: e1 (string)
        1: e2 (integer)
        2: e3 (boolean)
      + Cost Estimates:Estimated Node Cardinality: -1.0
      + Child 0:
        JoinNode
          + Output Columns:
            0: e1 (string)
            1: e1 (string)
            2: e3 (boolean)
          + Cost Estimates:Estimated Node Cardinality: -1.0
          + Child 0:
            AccessNode
              + Output Columns:e1 (string)
              + Cost Estimates:Estimated Node Cardinality: -1.0
              + Query:SELECT g_0.e1 AS c_0 FROM pm1.g1 AS g_0 ORDER BY c_0
              + Model Name:pm1
          + Child 1:
            AccessNode
              + Output Columns:
                0: e1 (string)
                1: e3 (boolean)
              + Cost Estimates:Estimated Node Cardinality: -1.0
              + Query:SELECT g_0.e1 AS c_0, g_0.e3 AS c_1 FROM pm1.g3 AS g_0 ORDER BY c_0
              + Model Name:pm1
          + Join Strategy:MERGE JOIN (ALREADY_SORTED/ALREADY_SORTED)
          + Join Type:INNER JOIN
          + Join Criteria:pm1.g1.e1=pm1.g3.e1
      + Child 1:
        AccessNode
          + Output Columns:
            0: e1 (string)
            1: e2 (integer)
          + Cost Estimates:Estimated Node Cardinality: -1.0
          + Query:SELECT g_0.e1 AS c_0, g_0.e2 AS c_1 FROM pm1.g2 AS g_0 ORDER BY c_0
          + Model Name:pm1
      + Join Strategy:ENHANCED SORT JOIN (SORT/ALREADY_SORTED)
      + Join Type:INNER JOIN
      + Join Criteria:pm1.g3.e1=pm1.g2.e1
  + Select Columns:
    0: pm1.g1.e1
    1: pm1.g2.e2
    2: pm1.g3.e3
```

Note that the nested join node is using a merge join and expects the source queries from each side to produce the expected ordering for the join. The parent join is an enhanced sort join which can delay the decision to perform sorting based upon the incoming rows. Note that the outer join from the user query has been modified to an inner join since none of the null inner values can be present in the query result.

The preceding plan can also be represented in in XML format as in the following example:

```
<?xml version="1.0" encoding="UTF-8"?>
<node name="ProjectNode">
    <property name="Output Columns">
        <value>e1 (string)</value>
        <value>e2 (integer)</value>
        <value>e3 (boolean)</value>
    </property>
    <property name="Cost Estimates">
        <value>Estimated Node Cardinality: -1.0</value>
    </property>
    <property name="Child 0">
        <node name="JoinNode">
            <property name="Output Columns">
                <value>e1 (string)</value>
                <value>e2 (integer)</value>
                <value>e3 (boolean)</value>
            </property>
            <property name="Cost Estimates">
                <value>Estimated Node Cardinality: -1.0</value>
            </property>
            <property name="Child 0">
                <node name="JoinNode">
                    <property name="Output Columns">
                        <value>e1 (string)</value>
                        <value>e1 (string)</value>
                        <value>e3 (boolean)</value>
                    </property>
                    <property name="Cost Estimates">
                        <value>Estimated Node Cardinality: -1.0</value>
                    </property>
                    <property name="Child 0">
                        <node name="AccessNode">
                            <property name="Output Columns">
                                <value>e1 (string)</value>
                            </property>
                            <property name="Cost Estimates">
                                <value>Estimated Node Cardinality: -1.0</value>
                            </property>
                            <property name="Query">
                                <value>SELECT g_0.e1 AS c_0 FROM pm1.g1 AS g_0 ORDER BY c_0</value>
                            </property>
                            <property name="Model Name">
                                <value>pm1</value>
                            </property>
                        </node>
                    </property>
                    <property name="Child 1">
                        <node name="AccessNode">
                            <property name="Output Columns">
                                <value>e1 (string)</value>
                                <value>e3 (boolean)</value>
                            </property>
                            <property name="Cost Estimates">
                                <value>Estimated Node Cardinality: -1.0</value>
                            </property>
                            <property name="Query">
                                <value>SELECT g_0.e1 AS c_0, g_0.e3 AS c_1 FROM pm1.g3 AS g_0
                                    ORDER BY c_0</value>
                            </property>
                            <property name="Model Name">
```

```
                                <value>pm1</value>
                            </property>
                        </node>
                    </property>
                    <property name="Join Strategy">
                        <value>MERGE JOIN (ALREADY_SORTED/ALREADY_SORTED)</value>
                    </property>
                    <property name="Join Type">
                        <value>INNER JOIN</value>
                    </property>
                    <property name="Join Criteria">
                        <value>pm1.g1.e1=pm1.g3.e1</value>
                    </property>
                </node>
            </property>
            <property name="Child 1">
                <node name="AccessNode">
                    <property name="Output Columns">
                        <value>e1 (string)</value>
                        <value>e2 (integer)</value>
                    </property>
                    <property name="Cost Estimates">
                        <value>Estimated Node Cardinality: -1.0</value>
                    </property>
                    <property name="Query">
                        <value>SELECT g_0.e1 AS c_0, g_0.e2 AS c_1 FROM pm1.g2 AS g_0
                            ORDER BY c_0</value>
                    </property>
                    <property name="Model Name">
                        <value>pm1</value>
                    </property>
                </node>
            </property>
            <property name="Join Strategy">
                <value>ENHANCED SORT JOIN (SORT/ALREADY_SORTED)</value>
            </property>
            <property name="Join Type">
                <value>INNER JOIN</value>
            </property>
            <property name="Join Criteria">
                <value>pm1.g3.e1=pm1.g2.e1</value>
            </property>
        </node>
    </property>
    <property name="Select Columns">
        <value>pm1.g1.e1</value>
        <value>pm1.g2.e2</value>
        <value>pm1.g3.e3</value>
    </property>
</node>
```

Note that the same information appears in each of the plan forms. In some cases it can actually be easier to follow the simplified format of the debug plan final processor plan. The following output shows how the debug log represents the plan in the preceding XML example:

```
OPTIMIZATION COMPLETE:
PROCESSOR PLAN:
ProjectNode(0) output=[pm1.g1.e1, pm1.g2.e2, pm1.g3.e3] [pm1.g1.e1, pm1.g2.e2, pm1.g3.e3]
  JoinNode(1) [ENHANCED SORT JOIN (SORT/ALREADY_SORTED)] [INNER JOIN] criteria=[pm1.g3.e1=pm1.g2.e1] output=[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
```

Common

- Output columns - what columns make up the tuples returned by this node.
- Data bytes sent - how many data byte, not including messaging overhead, were sent by this query.
- Planning time - the amount of time in milliseconds spent planning the query.

Relational

- Relational node ID — Matches the node ids seen in the debug log Node(id).
- Criteria — The Boolean expression used for filtering.
- Select columns — he columns that define the projection.
- Grouping columns — The columns used for grouping.
- Grouping mapping — Shows the mapping of aggregate and grouping column internal names to their expression form.
- Query — The source query.
- Model name — The model name.
- Sharing ID — Nodes sharing the same source results will have the same sharing id.
- Dependent join — If a dependent join is being used.
- Join strategy — The join strategy (Nested Loop, Sort Merge, Enhanced Sort, and so forth).
- Join type — The join type (Left Outer Join, Inner Join, Cross Join).
- Join criteria — The join predicates.
- Execution plan — The nested execution plan.
- Into target — The insertion target.
- Upsert — If the insert is an upsert.
- Sort columns — The columns for sorting.
- Sort mode — If the sort performs another function as well, such as distinct removal.
- Rollup — If the group by has the rollup option.
- Statistics — The processing statistics.
- Cost estimates — The cost/cardinality estimates including dependent join cost estimates.
- Row offset — The row offset expression.
- Row limit — The row limit expression.
- With — The with clause.
- Window functions — The window functions being computed.
- Table function — The table function (XMLTABLE, OBJECTTABLE, TEXTTABLE, and so forth).
- Streaming — If the XMLTABLE is using stream processing.

Procedure

- Expression
- Result Set

- Program
- Variable
- Then
- Else

Other plans

Procedure execution (including instead of triggers) use intermediate and final plan forms that include relational plans. Generally the structure of the XML/procedure plans will closely match their logical forms. It's the nested relational plans that will be of interest when analyzing performance issues.

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, and so forth, as possible. In many cases, such as with join ordering, planning is a combination of standard relational techniques along with costing information, and heuristics for pushdown optimization.

Criteria and join push down are typically the most important aspects of the query to push down when performance is a concern. For information about how to read a plan to ensure that source queries are as efficient as possible, see Query plans.

Dependent joins
A special optimization called a dependent join is used to reduce the rows returned from one of the two relations involved in a multi-source join. In a dependent join, queries are issued to each source sequentially rather than in parallel, with the results obtained from the first source used to restrict the records returned from the second. Dependent joins can perform some joins much faster by drastically reducing the amount of data retrieved from the second source and the number of join comparisons that must be performed.

The conditions when a dependent join is used are determined by the query planner based on access patterns, hints, and costing information. You can use the following types of dependent joins with Teiid:

Join based on equality or inequality
Where the engine determines how to break up large queries based on translator capabilities.

Key pushdown
Where the translator has access to the full set of key values and determines what queries to send.

Full pushdown
Where the translator ships the all data from the independent side to the translator. Can be used automatically by costing or can be specified as an option in the hint.

You can use the following types of hints in Teiid to control dependent join behavior:

MAKEIND
Indicates that the clause should be the independent side of a dependent join.

MAKEDEP
Indicates that the clause should be the dependent side of a join. As a non-comment hint, you can use `MAKEDEP` with the following optional `MAX` and `JOIN` arguments.

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

MAKEDEP(MAX:5000)

meaning that the dependent join should only be performed if there are less than the maximum number of values from the independent side.

MAKENOTDEP

Prevents the clause from being the dependent side of a join.

These can be placed in either the OPTION Clause or directly in the FROM Clause. As long as all Access Patterns can be met, the MAKEIND, MAKEDEP, and MAKENOTDEP hints override any use of costing information. MAKENOTDEP supersedes the other hints.

Tip	The MAKEDEP or MAKEIND hints should only be used if the proper query plan is not chosen by default. You should ensure that your costing information is representative of the actual source cardinality.

Note	An inappropriate MAKEDEP or MAKEIND hint can force an inefficient join structure and may result in many source queries.

Tip	While these hints can be applied to views, the optimizer will by default remove views when possible. This can result in the hint placement being significantly different than the original intention. You should consider using the NO_UNNEST hint to prevent the optimizer from removing the view in these cases.

In the simplest scenario the engine will use IN clauses (or just equality predicates) to filter the values coming from the dependent side. If the number of values from the independent side exceeds the translators `MaxInCriteriaSize` , the values will be split into multiple IN predicates up to `MaxDependentPredicates` . When the number of independent values exceeds `MaxInCriteriaSize*MaxDependentPredicates` , then multiple dependent queries will be issued in parallel.

If the translator returns true for `supportsDependentJoins` , then the engine may provide the entire set of independent key values. This will occur when the number of independent values exceeds `MaxInCriteriaSize*MaxDependentPredicates` so that the translator may use specific logic to avoid issuing multiple queries as would happen in the simple scenario.

If the translator returns true for both `supportsDependentJoins` and `supportsFullDependentJoins` then a full pushdown may be chosen by the optimizer A full pushdown, sometimes also called as data-ship pushdown, is where all the data from independent side of the join is sent to dependent side. This has an added benefit of allowing the plan above the join to be eligible for pushdown as well. This is why the optimizer may choose to perform a full pushdown even when the number of independent values does not exceed `MaxInCriteriaSize*MaxDependentPredicates` . You may also force full pushdown using the `MAKEDEP(JOIN)` hint. The translator is typically responsible for creating, populating, and removing a temporary table that represents the independent side. For more information about how to use custom translators with dependent, key, and full pushdown, see *Dependent join pushdown* in Translator Development. NOTE: By default, **Key/Full Pushdown** is compatible with only a subset of JDBC translators. To use it, set the translator override property `enableDependentJoins` to `true` . The JDBC source must allow for the creation of temporary tables, which typically requires a Hibernate dialect. The following translators are compatible with this feature: DB2, Derby, H2, Hana, HSQL, MySQL, Oracle, PostgreSQL, SQL Server, SAP IQ, Sybase, Teiid, and Teradata.

Copy criteria
Copy criteria is an optimization that creates additional predicates based upon combining join and where clause criteria. For example, equi-join predicates `(source1.table.column = source2.table.column)` are used to create new predicates by substituting `source1.table.column` for `source2.table.column` and vice versa. In a cross-source scenario, this allows for where criteria applied to a single side of the join to be applied to both source queries

Projection minimization
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, for example, `SELECT column3 FROM X` will the join be removed.

The optional join hint will not remove a bridging table that is still required.

Example: Bridging table

```
select a.column1, b.column2, c.column3 from /*+ optional */ a, b, c WHERE ON a.key = b.key AND a.key = c.key
```

Suppose this example defines a view layer `X` . If `b.column2` or `c.column3` are solely required by a query to `X` , then the join on a be removed. However if `a.column1` or both `b.column2` and `c.column3` are needed, then the optional join hint will not take effect.

When a join clause is omitted via the optional join hint, the relevant criteria is not applied. Thus it is possible that the query results may not have the same cardinality or even the same row values as when the join is fully applied.

Left/right outer joins where the inner side values are not used and whose rows under go a distinct operation will automatically be treated as an optional join and do not require a hint.

Example: Unnecessary optional join hint

```
    select distinct a.column1 from a LEFT OUTER JOIN /*+optional*/ b ON a.key = b.key
```

Note	A simple "SELECT COUNT(*) FROM VIEW" against a view where all join tables are marked as optional will not return a meaningful result.

Source hints

Teiid user and transformation queries can contain a meta source hint that can provide additional information to source queries. The source hint has the following form:

```
/*+ sh[[ KEEP ALIASES]:'arg'] source-name[ KEEP ALIASES]:'arg1' ... */
```

- The source hint is expected to appear after the query (SELECT, INSERT, UPDATE, DELETE) keyword.
- Source hints can appear in any subquery, or in views. All hints applicable to a given source query will be collected and pushed down together as a list. The order of the hints is not guaranteed.
- The sh arg is optional and is passed to all source queries via the `ExecutionContext.getGeneralHints` method. The additional args should have a source-name that matches the source name assigned to the translator in the VDB. If the source-name matches, the hint values will be supplied via the `ExecutionContext.getSourceHints` method. For more information about using an ExecutionContext, see Translator Development .
- Each of the arg values has the form of a string literal - it must be surrounded in single quotes and a single quote can be escaped with another single quote. Only the Oracle translator does anything with source hints by default. The Oracle translator will use both the source hint and the general hint (in that order) if available to form an Oracle hint enclosed in `/*+ … */`.
- If the KEEP ALIASES option is used either for the general hint or on the applicable source specific hint, then the table/view aliases from the user query and any nested views will be preserved in the push-down query. This is useful in situations where the source hint may need to reference aliases and the user does not wish to rely on the generated aliases (which can be seen in the query plan in the relevant source queries — see above). However in some situations this may result in an invalid source query if the preserved alias names are not valid for the source or result in a name collision. If the usage of KEEP ALIASES results in an error, the query could be modified by preventing view removal with the NO_UNNEST hint, the aliases modified, or the KEEP ALIASES option could be removed and the query plan used to determine the generated alias names.

Sample source hints

```
SELECT /*+ sh:'general hint' */ ...

SELECT /*+ sh KEEP ALIASES:'general hint' my-oracle:'oracle hint' */ ...
```

Partitioned union

Union partitioning is inferred from the transformation/inline view. If one (or more) of the UNION columns is defined by constants and/or has WHERE clause IN predicates containing only constants that make each branch mutually exclusive, then the UNION is considered partitioned. UNION ALL must be used and the UNION cannot have a LIMIT, WITH, or ORDER BY clause (although individual branches may use LIMIT, WITH, or ORDER BY). Partitioning values should not be null.

Example: Partitioned union

```
create view part as select 1 as x, y from foo union all select z, a from foo1 where z in (2, 3)
```

The view is partitioned on column x, since the first branch can only be the value 1 and the second branch can only be the values 2 or 3.

Note	More advanced or explicit partitioning will be considered for future releases.

The concept of a partitioned union is used for performing partition-wise joins, in Updatable Views, and Partial Aggregate Pushdown. These optimizations are also applied when using the multi-source feature as well - which introduces an explicit partitioning column.

Partition-wise joins take a join of unions and convert the plan into a union of joins, such that only matching partitions are joined against one another. If you want a partition-wise join to be performed implicit without the need for an explicit join predicate on the partitioning column, set the model/schema property `implicit_partition.columnName` to name of the partitioning column used on each partitioned view in the model/schema.

```
CREATE VIRTUAL SCHEMA all_customers SERVER server OPTIONS ("implicit_partition.columnName" 'theColumn');
```

In an XML VDB:

```
<?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 use with key joins. In some circumstances foreign key relationships on the same source should not be traversed at all or with the referenced table on the outer side of join. The extension property `teiid_rel:allow-join` can be used on the foreign key to further restrict the pushdown behavior. With a value of "false" no join pushdown will be allowed, and with a value of "inner" the referenced table must be on the inner side of the join. If the join pushdown is prevented, the join will be processed as a federated join.

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 merge join (MJ) hint directs the optimizer to use a traditional, semijoin, or antisemijoin merge join if possible. The dependent join (DJ) is the same as the MJ hint, but additionally directs the optimizer to use the subquery as the independent side of a dependent join if possible. This will only happen if the affected table has a primary key. If it does not, then an exception will be thrown.
- WHERE or HAVING clause IN, Quantified Comparison, Scalar Subquery Compare, and EXISTs predicates can take the MJ, DJ, or NO_UNNEST (no unnest) hints appearing just before the subquery. The NO_UNNEST hint, which supersedes the other hints, will direct the optimizer to leave the subquery in place.
- SELECT scalar subqueries can take the MJ or NO_UNNEST hints appearing just before the subquery. The MJ hint directs the optimizer to use a traditional or semijoin merge join if possible. The NO_UNNEST hint, which supersedes the other hints, will direct the optimizer to leave the subquery in place.

Merge join hint usage

```
SELECT col1 from tbl where col2 IN /*+ MJ*/ (SELECT col1 FROM tbl2)
```

Dependent join hint usage

```
SELECT col1 from tbl where col2 IN /*+ DJ */ (SELECT col1 FROM tbl2)
```

No unnest hint usage

```
SELECT col1 from tbl where col2 IN /*+ NO_UNNEST */ (SELECT col1 FROM tbl2)
```

- The system property `org.teiid.subqueryUnnestDefault` controls whether the optimizer will by default unnest subqueries during rewrite. If `true` , then most non-negated WHERE or HAVING clause EXISTS or IN subquery predicates can be converted to a traditional join.
- The planner will always convert to antijoin or semijoin variants if costing is favorable. Use a hint to override this behavior needed.
- EXISTs and scalar subqueries that are not pushed down, and not converted to merge joins, are implicitly limited to 1 and 2 result rows respectively via a limit.
- Conversion of subquery predicates to nested loop joins is not yet available.

XQuery optimization

A technique known as document projection is used to reduce the memory footprint of the context item document. Document projection loads only the parts of the document needed by the relevant XQuery and path expressions. Since document projection analysis uses all relevant path expressions, even 1 expression that could potentially use many nodes, for example, `//x rather than /a/b/x` will cause a larger memory footprint. With the relevant content removed the entire document will still be loaded into memory for processing. Document projection will only be used when there is a context item (unnamed PASSING clause item) passed to XMLTABLE/XMLQUERY. A named variable will not have document projection performed. In some cases the expressions used may be too complex for the optimizer to use document projection. You should check the SHOWPLAN DEBUG full plan output to see if the appropriate optimization has been performed.

With additional restrictions, simple context path expressions allow the processor to evaluate document subtrees independently - without loading the full document in memory. A simple context path expression can be of the form `[/][ns:]root/[ns1:]elem/…` `` ` ``, where a namespace prefix or element name can also be the * wild card. As with normal XQuery processing if namespace prefixes are used in the XQuery expression, they should be declared using the XMLNAMESPACES clause.

Streaming eligible XMLQUERY

```
XMLQUERY('/*:root/*:child' PASSING doc)
```

Rather than loading the entire doc in-memory as a DOM tree, each child element will be independently added to the result.

Streaming ineligible XMLQUERY

```
XMLQUERY('//child' PASSING doc)
```

The use of the descendant axis prevents the streaming optimization, but document projection can still be performed.

When using XMLTABLE, the COLUMN PATH expressions have additional restrictions. They are allowed to reference any part of the element subtree formed by the context expression and they may use any attribute value from their direct parentage. Any path expression where it is possible to reference a non-direct ancestor or sibling of the current context item prevent streaming from being used.

Streaming eligible XMLTABLE

```
XMLTABLE('/*:root/*:child' PASSING doc COLUMNS fullchild XML PATH '.', parent_attr string PATH '../@attr', chil
d_val integer)
```

The context XQuery and the column path expression allow the streaming optimization, rather than loading the entire document in-memory as a DOM tree, each child element will be independently added to the result.

Streaming ineligible XMLTABLE

```
XMLTABLE('/*:root/*:child' PASSING doc COLUMNS sibling_attr string PATH '../other_child/@attr')
```

The reference of an element outside of the child subtree in the sibling_attr path prevents the streaming optimization from being used, but document projection can still be performed.

Note	Column paths should be as targeted as possible to avoid performance issues. A general path such as `..//child` will cause the entire subtree of the context item to be searched on each output row.

Federated failure modes

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 that is associated with the source generates an exception in response to a query. The exception will be propagated to the query processor, where it will become a warning on the statement. For more information about partial results and SQLWarning objects, see *Partial Results Mode* in the Client Developer's Guide.

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 following extension metadata property to the appropriate source tables:

```
{http://www.teiid.org/ext/relational/2012}conformed-sources
```

You can set extension properties in the DDL file by using full DDL metadata or alter statements, or at runtime by using the `setProperty` system procedure. The property is expected to be a comma separated list of physical model/schema names.

DDL alter example

```
ALTER FOREIGN TABLE "reference_data" OPTIONS (ADD "teiid_rel:conformed-sources" 'source2,source3');
```

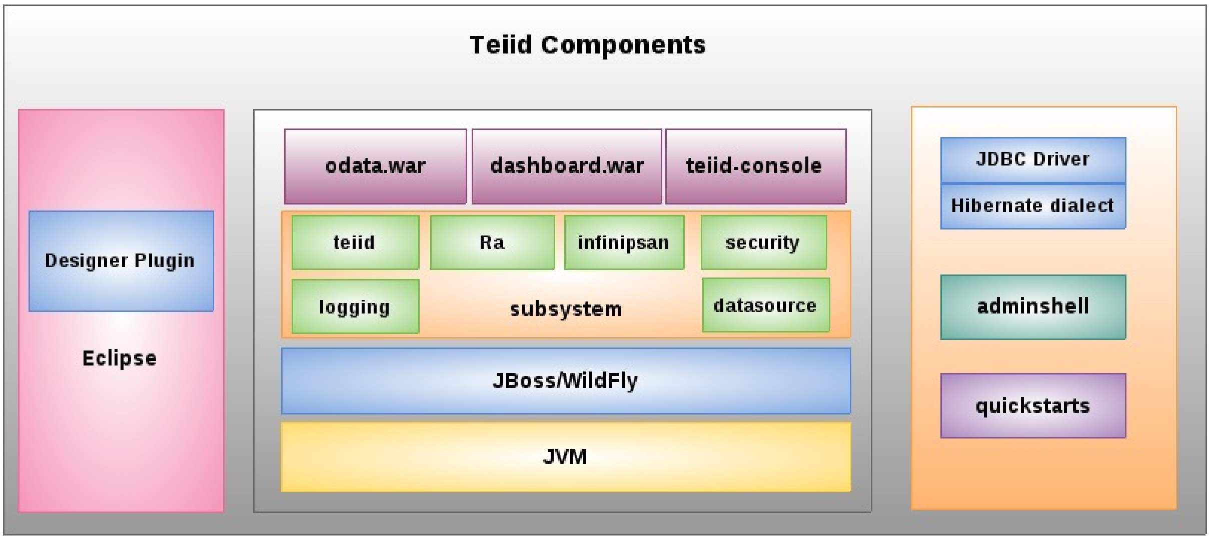
There is no expectation that a metadata entry exists on the other schemas.

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.

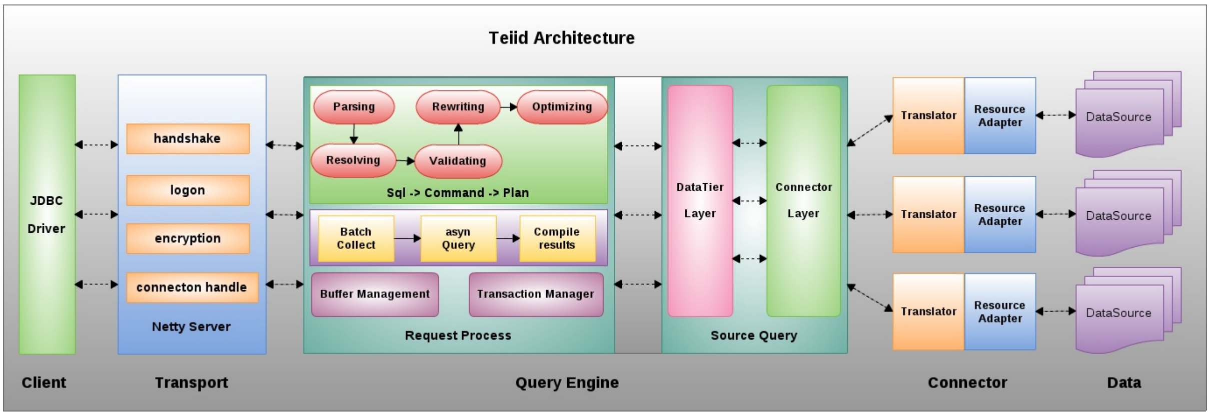
The engine will take the list of conformed sources and associate a set of model metadata ids to the corresponding access node. The logic considering joins and subqueries will also consider the conformed sets when making pushdown decisions. The subquery handling will only check for conformed sources for the subquery — not in the parent. So having a conformed table in the subquery will pushdown as expected, but not vice versa.

Teiid Architecture

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. For more information, see *OData support* in the Client Developer's Guide.
- **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.



- Client:: Client Developer's Guide

Transport

Transport services manage client connections: security authentication, encryption, and so forth.

Query Engine

The query engine has several layers and components. At a high level, request processing is structured as follows:

1. SQL is converted to a processor plan. The engine receives an incoming SQL query. It is parsed to a internal command. Then the command is converted a logical plan via resolving, validating, and rewriting. Finally, rule and cost-based optimization convert the logical plan to a final processor plan. For more information, see Federated planning.

2. Batch processing. The source and other aspects of query processing may return results asynchronously to the processing thread. As soon as possible, batches of results are made available to the client.

3. Buffer management controls the bulk of the on and off heap memory that 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 to interact directly with a source. Connectivity is provided for heterogeneous data stores, such as databases or data warehouses, NoSQL, Hadoop, data grid/cache, files, SaaS, and so on.

Translator

Teiid has developed a series of translators. For more information, see Translators.

Resource adapter

Provides container managed access to a source. For more information, see *Developing JEE connectors* in the Developer's Guide.

Terminology

VM or Process
A WildFly instance running Teiid.

Host
A machine that runs one or more VMs.

Service
A subsystem that runs in a VM (often in many VMs) and provides a related set of functionality. In addition to these main components, the service platform makes the following core set of services available to the applications that are built on top of it:

Session
The Session service manages active session information.

Buffer manager
The Buffer Manager service provides access to data management for intermediate results. For more information, see *Buffer management* in Data management.

Transaction
The Transaction service manages global, local, and request scoped transactions. For more information, see Transactions.

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, and so forth.) 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 caps the maximum number of open files to prevent running out of file handles. In cases with heavy buffering, this can cause wait times while waiting for a file handle to become available (the default maximum open files is 64).

Cleanup
When a tuple source is no longer needed, it is removed from the buffer manager. The buffer manager will remove it from both the memory storage manager and the disk storage manager. The disk storage manager will delete the file. In addition, every tuple source is tagged with a "group name" which is typically the session ID of the client. When the client’s session is terminated (by closing the connection, server detecting client shutdown, or administrative termination), a call is sent to the buffer manager to remove all tuple sources for the session.

In addition, when the query engine is shutdown, the buffer manager is shut down, which will remove all state from the disk storage manager and cause all files to be closed. When the query engine is stopped, it is safe to delete any files in the buffer directory as they are not used across query engine restarts and must be due to a system crash where buffer files were not cleaned up.

Query termination

Canceling queries
When a query is canceled, processing will be stopped in the query engine and in all connectors involved in the query. The semantics of what a connector does in response to a cancellation command depends on the connector implementation. For example, JDBC connectors will asynchronously call `cancel` on the underlying JDBC driver, which might or might not be compatible with this method.

User query timeouts
User query timeouts in 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 can also set a default statement timeout via the connection property "QUERYTIMEOUT". ODBC clients can also use QUERYTIMEOUT as an execution property via a set statement to control the default timeout setting. See the Client Developer’s Guide for more on connection/execution properties and set statements.

Server-side timeouts start when the query is received by the engine. Network latency or server load can delays the processing of I/O work after a client issues the query. The timeout will be cancelled if the first result is sent back before the timeout has ended. For more information about setting the `query-timeout` property for a virtual database, see Virtual database properties. For more information about modifying system properties to set a default query timeout for all queries, see *System properties* in the Administrator’s Guide.

Processing

Join algorithms
Nested loop does the most obvious processing. For every row in the outer source, it compares with every row in the inner source. Nested loop is only used when the join criteria has no equi-join predicates.

A merge join first sorts the input sources on the joined columns. You can then walk through each side in parallel (effectively one pass through each sorted source), and when you have a match, emit a row. In general, merge join is on the order of n+m rather than n*m in nested loop. Merge join is the default algorithm.

Using costing information the engine may also delay the decision to perform a full sort merge join. Based upon the actual row counts involved, the engine can choose to build an index of the smaller side (which will perform similarly to a hash join) or to only partially sort the larger side of the relation.

Joins involving equi-join predicates can be converted into dependent joins. For more information, see *Dependent joins* in Federated optimizations.

Sort-based algorithms
Sorting is used as the basis of the Sort (ORDER BY), Grouping (GROUP BY), and DupRemoval (SELECT DISTINCT) operations. The sort algorithm is a multi-pass merge-sort that does not require all of the result set to ever be in memory, yet uses the maximal amount of memory allowed by the buffer manager.

Sorting consists of two phases. In the first phase ("sort"), the algorithm processes an unsorted input stream to produce one or more sorted input streams. Each pass reads as much of the unsorted stream as possible, sorts it, and writes it back out as a new stream. When an unsorted stream is processed, the resulting sorted stream might be too large to reside in memory. If the size of a sorted stream exceeds the available memory, it is written out to multiple sorted streams.

The second phase of the sort algorithm ("merge") consists of a set of phases that grab the next batch from as many sorted input streams as will fit in memory. It then repeatedly grabs the next tuple in sorted order from each stream and outputs merged sorted batches to a new sorted stream. At completion of the phase, all input streams are dropped. In this way, each phase reduces the number of sorted streams. When only one stream remains, it is the final output.

BNF for SQL Grammar

Reserved Keywords

Keyword	Usage
ADD	add set child option, add set option, ADD column, ADD constraint
ALL	standard aggregate function, CREATE POLICY, function, GRANT, query expression body, query term, Revoke GRANT, select clause, quantified comparison predicate
ALTER	alter, ALTER PROCEDURE, alterStatement, ALTER TABLE, grant type
AND	between predicate, boolean term, window frame
ANY	standard aggregate function, with role, quantified comparison predicate
ARRAY	ARRAY expression constructor
ARRAY_AGG	ordered aggregate function
AS	alter, ALTER PROCEDURE, ALTER TABLE, ALTER TRIGGER, array table, create procedure, create a domain or type alias, option namespace, create trigger, create view, delete statement, derived column, dynamic data statement, function, json table, loop statement, xml namespace element, object table, select derived column, table subquery, text table, table name, unescapedFunction, update statement, with list element, xml serialize, xml table

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, GRANT, Revoke GRANT
COMMIT	create temporary table
CONSTRAINT	GRANT, table constraint
CONTINUE	branching statement
CONVERT	function
CREATE	create procedure, create data wrapper, create database, create a domain or type alias, create foreign temp table, CREATE POLICY, create role, create schema, create server, create table, create temporary table, create trigger

CROSS	cross join
CUME_DIST	analytic aggregate function
CURRENT_DATE	function
CURRENT_TIME	function
CURRENT_TIMESTAMP	function
DATE	non numeric literal, simple data type
DAY	function
DECIMAL	simple data type
DECLARE	declare statement
DELETE	alter, ALTER TRIGGER, CREATE POLICY, create trigger, delete statement, grant type
DESC	sort specification
DISTINCT	standard aggregate function, function, is distinct, query expression body, query term, select clause
DOUBLE	simple data type
DROP	DROP column, drop option, Drop data wrapper, drop option, DROP POLICY, drop procedure, drop role, drop schema, drop server, drop table, drop table, grant type
EACH	for each row trigger action
ELSE	case expression, if statement, searched case expression
END	case expression, compound statement, for each row trigger action, searched case expression
ERROR	raise error statement
ESCAPE	match predicate, text table
EXCEPT	query expression body
EXEC	dynamic data statement, call statement
EXECUTE	dynamic data statement, grant type, call statement
EXISTS	exists predicate
FALSE	explain option, json table, non numeric literal
FETCH	fetch clause

FETCH	fetch clause
FILTER	filter clause
FLOAT	simple data type
FOR	CREATE POLICY, for each row trigger action, function, json table column, text aggreate function, text table column, xml table column
FOREIGN	ALTER PROCEDURE, ALTER TABLE, create procedure, create data wrapper, create foreign or global temporary table, create foreign temp table, create schema, create server, Drop data wrapper, drop procedure, drop schema, drop table, foreign key, Import foreign schema, with role
FROM	delete statement, from clause, function, Import foreign schema, is distinct, Revoke GRANT
FULL	qualified table
FUNCTION	create procedure, drop procedure, GRANT, Revoke GRANT
GLOBAL	create foreign or global temporary table, drop table
GRANT	GRANT
GROUP	function, group by clause
HANDLER	create data wrapper
HAVING	having clause
HOUR	function
IF	if statement
IMMEDIATE	dynamic data statement
IMPORT	Import another Database, Import foreign schema
IN	function, procedure parameter, in predicate
INNER	qualified table
INOUT	procedure parameter
INSERT	alter, ALTER TRIGGER, CREATE POLICY, create trigger, function, insert statement, grant type
INTEGER	simple data type
INTERSECT	query term

INTO	dynamic data statement, Import foreign schema, insert statement, into clause
IS	is distinct, is null predicate
JOIN	cross join, make dep options, qualified table
LANGUAGE	GRANT, object table, Revoke GRANT
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, GRANT, 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
OF	alter, ALTER TRIGGER, create trigger
OFFSET	limit clause
ON	alter, ALTER TRIGGER, create foreign temp table, CREATE POLICY, create temporary table, create trigger, DROP POLICY, GRANT, loop statement, qualified table, Revoke GRANT, xml query
ONLY	fetch clause
OPTION	option clause
OPTIONS	alter child options list, alter options list, options clause
OR	boolean value expression
ORDER	GRANT, order by clause
OUT	procedure parameter
OUTER	qualified table
OVER	window specification
PARAMETER	ALTER PROCEDURE
PARTITION	window specification
PERCENT_RANK	analytic aggregate function
PRIMARY	create temporary table, inline constraint, primary key
PROCEDURE	alter, ALTER PROCEDURE, create procedure, CREATE POLICY, DROP POLICY, drop procedure, GRANT, Revoke GRANT
RANGE	window frame
REAL	simple data type
REFERENCES	foreign key
RETURN	assignment statement, return statement, data statement
RETURNS	create procedure
REVOKE	Revoke GRANT

RIGHT	function, qualified table
ROLLUP	group by clause
ROW	array table, fetch clause, for each row trigger action, limit clause, text table, window frame bound
ROWS	array table, create temporary table, fetch clause, limit clause, window frame
SECOND	function
SELECT	CREATE POLICY, grant type, select clause
SERVER	ALTER SERVER, create schema, create server, drop server, Import foreign schema
SET	add set child option, add set option, option namespace, update statement, set schema
SHORT	simple data type
SIMILAR	match predicate
SMALLINT	simple data type
SOME	standard aggregate function, quantified comparison predicate
SQLEXCEPTION	sql exception
SQLSTATE	sql exception
SQLWARNING	raise statement
STRING	dynamic data statement, simple data type, xml serialize
TABLE	ALTER TABLE, create procedure, create foreign or global temporary table, create foreign temp table, create temporary table, drop table, drop table, GRANT, query primary, Revoke GRANT, table subquery
TEMPORARY	create foreign or global temporary table, create foreign temp table, create temporary table, drop table, GRANT, Revoke GRANT
THEN	case expression, searched case expression
TIME	non numeric literal, simple data type
TIMESTAMP	non numeric literal, simple data type
TINYINT	simple data type

TO	rename column options, RENAME Table, CREATE POLICY, DROP POLICY, GRANT, match predicate
TRAILING	function
TRANSLATE	function
TRIGGER	alter, ALTER TRIGGER, create trigger
TRUE	explain option, json table, non numeric literal
UNION	cross join, query expression body
UNIQUE	other constraints, inline constraint
UNKNOWN	non numeric literal
UPDATE	alter, ALTER TRIGGER, CREATE POLICY, create trigger, dynamic data statement, grant type, update statement
USER	function
USING	CREATE POLICY, dynamic data statement
VALUES	query primary
VARBINARY	simple data type, xml serialize
VARCHAR	simple data type, xml serialize
VIRTUAL	ALTER PROCEDURE, ALTER TABLE, create procedure, create schema, create view, drop procedure, drop schema, drop table
WHEN	case expression, searched case expression
WHERE	filter clause, where clause
WHILE	while statement
WITH	assignment statement, create role, Import another Database, query expression, data statement
WITHIN	function
WITHOUT	assignment statement, data statement
WRAPPER	ALTER DATA WRAPPER, create data wrapper, create server, Drop data wrapper
XML	explain option, simple data type
XMLAGG	ordered aggregate function

XMLATTRIBUTES	xml attributes
XMLCAST	unescapedFunction
XMLCOMMENT	function
XMLCONCAT	function
XMLELEMENT	xml element
XMLEXISTS	xml query
XMLFOREST	xml forest
XMLNAMESPACES	xml namespaces
XMLPARSE	xml parse
XMLPI	function
XMLQUERY	xml query
XMLSERIALIZE	xml serialize
XMLTABLE	xml table
XMLTEXT	function
YEAR	function

Non-Reserved Keywords

Name	Usage
ACCESS	basicNonReserved, Import another Database
ACCESSPATTERN	basicNonReserved, other constraints
AFTER	alter, basicNonReserved, create trigger
ANALYZE	basicNonReserved, explain option
ARRAYTABLE	array table, basicNonReserved
AUTO_INCREMENT	alter column options, basicNonReserved, table element, view element
AVG	standard aggregate function, basicNonReserved
CHAIN	basicNonReserved, sql exception

COLUMNS	array table, basicNonReserved, json table, object table, text table, xml table
CONDITION	basicNonReserved, GRANT, Revoke GRANT
CONTENT	basicNonReserved, xml parse, xml serialize
CONTROL	basicNonReserved, Import another Database
COUNT	standard aggregate function, basicNonReserved
COUNT_BIG	standard aggregate function, basicNonReserved
CURRENT	basicNonReserved, window frame bound
DATA	ALTER DATA WRAPPER, basicNonReserved, create data wrapper, create server, Drop data wrapper
DATABASE	ALTER DATABASE, basicNonReserved, create database, Import another Database, use database
DEFAULT	xml namespace element, non-reserved identifier, object table column, post create column, procedure parameter, xml table column
DELIMITER	basicNonReserved, text aggreate function, text table
DENSE_RANK	analytic aggregate function, basicNonReserved
DISABLED	alter, ALTER TRIGGER, basicNonReserved
DOCUMENT	basicNonReserved, xml parse, xml serialize
DOMAIN	basicNonReserved, create a domain or type alias
EMPTY	basicNonReserved, xml query
ENABLED	alter, ALTER TRIGGER, basicNonReserved
ENCODING	basicNonReserved, text aggreate function, xml serialize
EPOCH	basicNonReserved, function
EVERY	standard aggregate function, basicNonReserved
EXCEPTION	compound statement, declare statement, non-reserved identifier
EXCLUDING	basicNonReserved, xml serialize
EXPLAIN	basicNonReserved, explain
EXTRACT	basicNonReserved, function

FIRST	basicNonReserved, fetch clause, sort specification
FOLLOWING	basicNonReserved, window frame bound
FORMAT	basicNonReserved, explain option
GEOGRAPHY	non-reserved identifier, simple data type
GEOMETRY	non-reserved identifier, simple data type
HEADER	basicNonReserved, text aggreate function, text table column, text table
INCLUDING	basicNonReserved, xml serialize
INDEX	other constraints, inline constraint, non-reserved identifier
INSTEAD	alter, ALTER TRIGGER, basicNonReserved, create trigger
JAAS	basicNonReserved, with role
JSON	non-reserved identifier, simple data type
JSONARRAY_AGG	basicNonReserved, ordered aggregate function
JSONOBJECT	basicNonReserved, json object
JSONTABLE	basicNonReserved, json table
KEY	basicNonReserved, create temporary table, foreign key, inline constraint, primary key
LAST	basicNonReserved, sort specification
LISTAGG	basicNonReserved, function
MASK	basicNonReserved, GRANT, Revoke GRANT
MAX	standard aggregate function, basicNonReserved, make dep options
MIN	standard aggregate function, basicNonReserved
NAME	basicNonReserved, function, xml element
NAMESPACE	basicNonReserved, option namespace
NEXT	basicNonReserved, fetch clause
NONE	basicNonReserved
NULLS	basicNonReserved, sort specification

OBJECT	non-reserved identifier, simple data type
OBJECTTABLE	basicNonReserved, object table
ORDINALITY	basicNonReserved, json table column, text table column, xml table column
PASSING	basicNonReserved, object table, xml query, xml query, xml table
PATH	basicNonReserved, json table column, xml table column
POLICY	basicNonReserved, CREATE POLICY, DROP POLICY
POSITION	basicNonReserved, function
PRECEDING	basicNonReserved, window frame bound
PRESERVE	basicNonReserved, create temporary table
PRIVILEGES	basicNonReserved, GRANT, Revoke GRANT
QUARTER	basicNonReserved, function
QUERYSTRING	basicNonReserved, querystring function
QUOTE	basicNonReserved, text aggreate function, text table
RAISE	basicNonReserved, raise statement
RANK	analytic aggregate function, basicNonReserved
RENAME	ALTER PROCEDURE, ALTER TABLE, basicNonReserved
REPOSITORY	basicNonReserved, Import foreign schema
RESULT	basicNonReserved, procedure parameter
ROLE	basicNonReserved, create role, drop role, with role
ROW_NUMBER	analytic aggregate function, basicNonReserved
SCHEMA	basicNonReserved, create schema, drop schema, GRANT, Import foreign schema, Revoke GRANT, set schema
SELECTOR	basicNonReserved, text table column, text table
SERIAL	alter column options, table element, view element, non-reserved identifier, temporary table element
SKIP	basicNonReserved, text table

SQL_TSI_DAY	basicNonReserved, time interval
SQL_TSI_FRAC_SECOND	basicNonReserved, time interval
SQL_TSI_HOUR	basicNonReserved, time interval
SQL_TSI_MINUTE	basicNonReserved, time interval
SQL_TSI_MONTH	basicNonReserved, time interval
SQL_TSI_QUARTER	basicNonReserved, time interval
SQL_TSI_SECOND	basicNonReserved, time interval
SQL_TSI_WEEK	basicNonReserved, time interval
SQL_TSI_YEAR	basicNonReserved, time interval
STDDEV_POP	standard aggregate function, basicNonReserved
STDDEV_SAMP	standard aggregate function, basicNonReserved
SUBSTRING	basicNonReserved, function
SUM	standard aggregate function, basicNonReserved
TEXT	basicNonReserved, explain option
TEXTAGG	basicNonReserved, text aggreate function
TEXTTABLE	basicNonReserved, text table
TIMESTAMPADD	basicNonReserved, function
TIMESTAMPDIFF	basicNonReserved, function
TO_BYTES	basicNonReserved, function
TO_CHARS	basicNonReserved, function
TRANSLATOR	ALTER DATA WRAPPER, basicNonReserved, create data wrapper, create server, Drop data wrapper
TRIM	basicNonReserved, function, text table column, text table
TYPE	alter column options, basicNonReserved, create data wrapper, create server
UNBOUNDED	basicNonReserved, window frame bound
UPSERT	basicNonReserved, insert statement
USAGE	basicNonReserved, GRANT, Revoke GRANT

USE	basicNonReserved, use database
VARIADIC	basicNonReserved, procedure parameter
VAR_POP	standard aggregate function, basicNonReserved
VAR_SAMP	standard aggregate function, basicNonReserved
VERSION	basicNonReserved, create database, create server, Import another Database, use database, xml serialize
VIEW	alter, ALTER TABLE, basicNonReserved, create view, drop table
WELLFORMED	basicNonReserved, xml parse
WIDTH	basicNonReserved, text table column
XMLDECLARATION	basicNonReserved, xml serialize
YAML	basicNonReserved, explain option

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

<table>
<tr><td>comma</td><td>","</td><td>alter child options list, alter options list, ARRAY expression constructor, column list, create procedure, typed element list, CREATE POLICY, create table body, create temporary table, create view body, derived column list, sql exception, named parameter list, explain, expression list, from clause, function, GRANT, identifier list, json table, limit clause, nested expression, object table, option clause, options clause, order by clause, simple data type, query expression, query primary, querystring function, Revoke GRANT, select clause, set clause list, in predicate, text aggreate function, text table, xml attributes, xml element, xml query, xml forest, xml namespaces, xml query, xml table</td></tr>
<tr><td>concat_op</td><td>"||"</td><td>common value expression</td></tr>
<tr><td>decimal numeric literal</td><td>(<digit>)* <period> <unsigned integer literal></td><td>unsigned numeric literal</td></tr>
<tr><td>digit</td><td>\["0"\-"9"\]</td><td></td></tr>
<tr><td>dollar</td><td>"$"</td><td>parameter reference</td></tr>
<tr><td>double_amp_op</td><td>"&&"</td><td>common value expression</td></tr>
<tr><td>eq</td><td>"="</td><td>assignment statement, callable statement, declare statement, named parameter list, comparison operator, set clause list</td></tr>
<tr><td>escaped function</td><td>"{" "fn"</td><td>unsigned value expression primary</td></tr>
<tr><td>escaped join</td><td>"{" "oj"</td><td>table reference</td></tr>
<tr><td>escaped type</td><td>"{" ("d" | "t" | "ts" | "b")</td><td>non numeric literal</td></tr>
<tr><td>approximate numeric literal</td><td><digit> <period> <unsigned integer literal> \["e","E"\] (<plus> | <minus>)? <unsigned integer literal></td><td>unsigned numeric literal</td></tr>
<tr><td>ge</td><td>">="</td><td>comparison operator</td></tr>
<tr><td>gt</td><td>">"</td><td>named parameter list, comparison operator</td></tr>
<tr><td>hexit</td><td>\["a"\-"f","A"\-"F"\] | <digit></td><td></td></tr>
<tr><td>identifier</td><td><quoted_id> (<period> <quoted_id>)*</td><td>create a domain or type alias, identifier, data type, Unqualified identifier, unsigned value expression primary</td></tr>
</table>

<table>
<tr><td>id_part</td><td>("" | "@" | "#" | <letter>) (<letter> | "" | <digit>)*</td><td></td></tr>
<tr><td>lbrace</td><td>"{"</td><td>callable statement, match predicate</td></tr>
<tr><td>le</td><td>"<="</td><td>comparison operator</td></tr>
<tr><td>letter</td><td>\["a"\-"z","A"\-"Z"\] | \["\u0153"\-"\ufffd"\]</td><td></td></tr>
<tr><td>lparen</td><td>"("</td><td>standard aggregate function, alter child options list, alter options list, analytic aggregate function, ARRAY expression constructor, array table, callable statement, column list, other constraints, create procedure, CREATE POLICY, create table body, create temporary table, create view body, explain, filter clause, function, group by clause, if statement, json object, json table, loop statement, make dep options, nested expression, object table, options clause, ordered aggregate function, simple data type, query primary, querystring function, in predicate, call statement, subquery, quantified comparison predicate, table subquery, table primary, text aggreate function, text table, unescapedFunction, while statement, window specification, with list element, xml attributes, xml element, xml query, xml forest, xml namespaces, xml parse, xml query, xml serialize, xml table</td></tr>
<tr><td>lsbrace</td><td>"["</td><td>ARRAY expression constructor, basic data type, data type, value expression primary</td></tr>
<tr><td>lt</td><td>"<"</td><td>comparison operator</td></tr>
<tr><td>minus</td><td>"-"</td><td>plus or minus</td></tr>
<tr><td>ne</td><td>"<>"</td><td>comparison operator</td></tr>
<tr><td>ne2</td><td>"!="</td><td>comparison operator</td></tr>
<tr><td>period</td><td>"."</td><td></td></tr>
<tr><td>plus</td><td>"+"</td><td>plus or minus</td></tr>
<tr><td>qmark</td><td>"?"</td><td>callable statement, parameter reference</td></tr>
<tr><td>quoted_id</td><td><id_part> | "\"" ("\"\"" | ~\["\""\])+ "\""</td><td></td></tr>
</table>

rbrace	"}"	callable statement, match predicate, non numeric literal, table reference, unsigned value expression primary
rparen	")"	standard aggregate function, alter child options list, alter options list, analytic aggregate function, ARRAY expression constructor, array table, callable statement, column list, other constraints, create procedure, CREATE POLICY, create table body, create temporary table, create view body, explain, filter clause, function, group by clause, if statement, json object, json table, loop statement, make dep options, nested expression, object table, options clause, ordered aggregate function, simple data type, query primary, querystring function, in predicate, call statement, subquery, quantified comparison predicate, table subquery, table primary, text aggregate 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
basicNonReserved	create a domain or type alias, non-reserved identifier, data type
between predicate	boolean primary

boolean primary	CREATE POLICY, filter clause, GRANT, boolean factor
branching statement	delimited statement
callable 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 POLICY	ddl statement
create role	ddl statement
create schema	ddl statement
create server	ddl statement

create table	ddl statement
create table body	create foreign or global temporary table, create foreign temp table
create temporary table	directly executable statement
create trigger	ddl statement, directly executable statement
create view	create table
create view body	create view
view element	create view body
condition	expression, having clause, if statement, qualified table, searched case expression, where clause, while statement
cross join	joined table
ddl statement	ddl statement
declare statement	delimited statement
delete statement	assignment statement operand, directly executable statement
delimited statement	statement
derived column	derived column list, object table, querystring function, text aggregate function, xml attributes, xml query, xml query, xml table
derived column list	json object, xml forest
drop option	alter child options list
Drop data wrapper	ddl statement
drop option	alter options list
DROP POLICY	ddl statement
drop procedure	ddl statement
drop role	ddl statement
drop schema	ddl statement
drop server	ddl statement
drop table	directly executable statement

drop table	ddl statement
dynamic data statement	data statement
raise error statement	delimited statement
sql exception	assignment statement operand, exception reference
exception reference	sql exception, raise statement
named parameter list	callable statement, call statement
exists predicate	boolean primary
explain	
explain option	explain
expression	standard aggregate function, ARRAY expression constructor, assignment statement operand, case expression, derived column, dynamic data statement, raise error statement, named parameter list, expression list, function, nested expression, object table column, ordered aggregate function, post create column, procedure parameter, querystring function, return statement, searched case expression, select derived column, set clause list, sort key, quantified comparison predicate, unescapedFunction, xml table column, xml element, xml parse, xml serialize
expression list	callable statement, other constraints, function, group by clause, query primary, call statement, window specification
fetch clause	limit clause
filter clause	function, unescapedFunction
for each row trigger action	alter, ALTER TRIGGER, create trigger
foreign key	table constraint
from clause	query
function	unescapedFunction, unsigned value expression primary
GRANT	ddl statement
group by clause	query
having clause	query
	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

identifier	list, compound statement, table element, create data wrapper, create database, typed element list, create foreign temp table, option namespace, CREATE POLICY, create schema, create trigger, view element, declare statement, delete statement, derived column, drop option, Drop data wrapper, drop option, DROP POLICY, drop procedure, drop role, drop schema, drop server, drop table, drop table, dynamic data statement, exception reference, named parameter list, foreign key, function, GRANT, identifier list, Import another Database, Import foreign schema, insert statement, into clause, json table column, json table, loop statement, xml namespace element, object table column, object table, option clause, option pair, procedure parameter, procedure result column, query primary, Revoke GRANT, select derived column, set clause list, statement, call statement, table subquery, table constraint, temporary table element, text aggregate function, text table column, text table, table name, update statement, use database, set schema, with list element, xml table column, xml element, xml serialize, xml table
identifier list	create schema, with role
if statement	statement
Import another Database	ddl statement
Import foreign schema	ddl statement
inline constraint	post create column
insert statement	assignment statement operand, directly executable statement
integer parameter	fetch clause, limit clause
unsigned integer	dynamic data statement, function, GRANT, integer parameter, make dep options, parameter reference, simple data type, text table column, text table, window frame bound
time interval	function
into clause	query
is distinct	boolean primary
is null predicate	boolean primary
joined table	table primary, table reference
json table column	json table
json object	function
json table	table primary
limit clause	query expression body

loop statement	statement
make dep options	option clause, table primary
match predicate	boolean primary
xml namespace element	xml namespaces
nested expression	unsigned value expression primary
non numeric literal	alter child option pair, alter option pair, option pair, value expression primary
non-reserved identifier	identifier, Unqualified identifier, unsigned value expression primary
boolean factor	boolean term
object table column	object table
object table	table primary
comparison operator	comparison predicate, quantified comparison predicate
option clause	callable statement, delete statement, insert statement, query expression body, call statement, update statement
option pair	options clause
options clause	create procedure, create data wrapper, create database, create schema, create server, create table body, create view, create view body, Import foreign schema, post create column, procedure parameter, procedure result column, table constraint
order by clause	function, ordered aggregate function, query expression body, text aggreate function, window specification
ordered aggregate function	unescapedFunction
parameter reference	unsigned value expression primary
basic data type	typed element list, json table column, object table column, data type, temporary table element, text table column, xml table column
data type	alter column options, table element, create procedure, create a domain or type alias, view element, declare statement, function, procedure parameter, procedure result column, unescapedFunction
simple data type	basic data type
numeric value expression	common value expression, value expression primary

plus or minus	alter child option pair, alter option pair, option pair, numeric value expression, value expression primary
post create column	table element, view element
primary key	table constraint
procedure parameter	create procedure
procedure result column	create procedure
qualified table	joined table
query	query primary
query expression	alter, ALTER TABLE, ARRAY expression constructor, assignment statement operand, create view, insert statement, loop statement, subquery, table subquery, directly executable statement, with list element
query expression body	query expression, query primary
query primary	query term
querystring function	function
query term	query expression body
raise statement	delimited statement
grant type	GRANT, Revoke GRANT
with role	create role
like regex predicate	boolean primary
return statement	delimited statement
Revoke GRANT	ddl statement
searched case expression	unsigned value expression primary
select clause	query
select derived column	select sublist
select sublist	select clause
set clause list	dynamic data statement, update statement
in predicate	boolean primary
sort key	sort specification

sort specification	order by clause
data statement	delimited statement
statement	alter, ALTER PROCEDURE, compound statement, create procedure, for each row trigger action, if statement, loop statement, while statement
call statement	assignment statement, subquery, table subquery, directly executable statement
string	character, create database, option namespace, create server, function, GRANT, Import another Database, json table column, json table, xml namespace element, non numeric literal, object table column, object table, text table column, text table, use database, xml table column, xml query, xml query, xml serialize, xml table
subquery	exists predicate, in predicate, quantified comparison predicate, unsigned value expression primary
quantified comparison predicate	boolean primary
table subquery	table primary
table constraint	ADD constraint, create table body, create view body
temporary table element	create temporary table
table primary	cross join, joined table
table reference	from clause, qualified table
text 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 foreign temp table, create role, create server, create temporary table, create view
unsigned numeric literal	alter child option pair, alter option pair, option pair, value expression primary

unsigned value expression primary	integer parameter, value expression primary
update statement	assignment statement operand, directly executable statement
use database	ddl statement
set schema	ddl statement
directly executable statement	explain, data statement
value expression primary	array table, json table, term
where clause	delete statement, query, update statement
while statement	statement
window frame	window specification
window frame bound	window frame
window specification	unescapedFunction
with list element	query expression
xml attributes	xml element
xml table column	xml table
xml element	function
xml query	boolean primary
xml forest	function
xml namespaces	xml element, xml query, xml forest, xml query, xml table
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 ::=

- EXCEPTION
- SERIAL
- OBJECT
- INDEX
- JSON
- GEOMETRY
- GEOGRAPHY
- DEFAULT
- <basicNonReserved>

Allows non-reserved keywords to be parsed as identifiers

Example: SELECT **COUNT** FROM …

basicNonReserved ::=

- INSTEAD
- VIEW
- ENABLED
- DISABLED
- KEY
- TEXTAGG
- COUNT
- COUNT_BIG
- ROW_NUMBER
- RANK
- DENSE_RANK
- SUM
- AVG

- MIN
- MAX
- EVERY
- STDDEV_POP
- STDDEV_SAMP
- VAR_SAMP
- VAR_POP
- DOCUMENT
- CONTENT
- TRIM
- EMPTY
- ORDINALITY
- PATH
- FIRST
- LAST
- NEXT
- SUBSTRING
- EXTRACT
- TO_CHARS
- TO_BYTES
- TIMESTAMPADD
- TIMESTAMPDIFF
- QUERYSTRING
- NAMESPACE
- RESULT
- ACCESSPATTERN
- AUTO_INCREMENT
- WELLFORMED
- SQL_TSI_FRAC_SECOND
- SQL_TSI_SECOND
- SQL_TSI_MINUTE
- SQL_TSI_HOUR
- SQL_TSI_DAY
- SQL_TSI_WEEK

- SQL_TSI_MONTH
- SQL_TSI_QUARTER
- SQL_TSI_YEAR
- TEXTTABLE
- ARRAYTABLE
- JSONTABLE
- SELECTOR
- SKIP
- WIDTH
- PASSING
- NAME
- ENCODING
- COLUMNS
- DELIMITER
- QUOTE
- HEADER
- NULLS
- OBJECTTABLE
- VERSION
- INCLUDING
- EXCLUDING
- XMLDECLARATION
- VARIADIC
- RAISE
- CHAIN
- JSONARRAY_AGG
- JSONOBJECT
- PRESERVE
- UPSERT
- AFTER
- TYPE
- TRANSLATOR
- JAAS
- CONDITION

- MASK
- ACCESS
- CONTROL
- NONE
- DATA
- DATABASE
- PRIVILEGES
- ROLE
- SCHEMA
- USE
- REPOSITORY
- RENAME
- DOMAIN
- USAGE
- POSITION
- CURRENT
- UNBOUNDED
- PRECEDING
- FOLLOWING
- LISTAGG
- EXPLAIN
- ANALYZE
- TEXT
- FORMAT
- YAML
- EPOCH
- QUARTER
- POLICY

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
```

---

explain ::=

- EXPLAIN (<lparen> <explain option> (<comma> <explain option>)* <rparen>)? <directly executable statement>

Returns the query plan for the statement

Example: EXPLAIN select 1

---

explain option ::=

- (ANALYZE (TRUE | FALSE)?)
- (FORMAT (XML | TEXT | YAML)?)

Option for the explain statement

Example: FORMAT YAML

---

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 <Unqualified identifier> <lparen> <temporary table element> (<comma> <temporary table element>)* (<comma> PRIMARY KEY <column list>)? <rparen> (ON COMMIT PRESERVE ROWS)?

Creates a temporary table.

Example:

```
CREATE LOCAL TEMPORARY TABLE tmp (col integer)
```

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.

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 aggregate 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> | <json table> | <xml table> | <object table> | <table name> | <table subquery> | (<lparen> <joined table> <rparen>)) ((MAKEDEP <make dep options>) | MAKENOTDEP)? ((MAKEIND <make dep options>))?

A single source of rows.

Example:

```
a
```

---

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> (ROW | ROWS)? <value expression primary> COLUMNS <typed element list> <rparen> (AS)? <identifier>

The ARRAYTABLE table function creates tabular results from arrays. It can be used as a nested table reference.

Example:

```
ARRAYTABLE (col1 COLUMNS x STRING) AS y
```

---

json table ::=

- JSONTABLE <lparen> <value expression primary> <comma> <string> (<comma> (TRUE | FALSE))? COLUMNS <json table column> (<comma> <json table column>)* <rparen> (AS)? <identifier>

The JSONTABLE table function creates tabular results from JSON. It can be used as a nested table reference.

Example:

```
JSONTABLE (col1, '$..book', false COLUMNS x STRING) AS y
```

---

json table column ::=

- <identifier> ((FOR ORDINALITY) | (<basic data type> (PATH <string>)?))

json table column.

Example:

```
col FOR ORDINALITY
```

---

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>) | (<lparen> <query expression> <rparen>))

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 | QUARTER | EPOCH) FROM <expression> <rparen>)
- (TRIM <lparen> ((((LEADING | TRAILING | BOTH) (<expression>)?) | <expression>) FROM)? <expression> <rparen>)
- ((TO_CHARS | TO_BYTES) <lparen> <expression> <comma> <string> (<comma> <expression>)? <rparen>)
- ((TIMESTAMPADD | TIMESTAMPDIFF) <lparen> <time interval> <comma> <expression> <comma> <expression> <rparen>)
- <querystring function>
- ((LEFT | RIGHT | CHAR | USER | YEAR | MONTH | HOUR | MINUTE | SECOND | XMLCONCAT | XMLCOMMENT | XMLTEXT) <lparen> (<expression list>)? <rparen>)
- ((TRANSLATE | INSERT) <lparen> (<expression list>)? <rparen>)
- <xml parse>
- <xml element>
- (XMLPI <lparen> ((NAME)? <identifier>) (<comma> <expression>)? <rparen>)
- <xml forest>
- <json object>

- <xml serialize>
- <xml query>
- (POSITION <lparen> <common value expression> IN <common value expression> <rparen>)
- (LISTAGG <lparen> <expression> (<comma> <string>)? <rparen> WITHIN GROUP <lparen> <order by clause> <rparen>)
- (<identifier> <lparen> (ALL | DISTINCT)? (<expression list>)? (<order by clause>)? <rparen> (<filter clause>)?)
- (CURRENT_DATE (<lparen> <rparen>)?)
- ((CURRENT_TIMESTAMP | CURRENT_TIME) (<lparen> <unsigned integer> <rparen>)?)

Calls a scalar function.

Example:

```
func('1', col1)
```

---

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 (<lparen> <unsigned integer> <rparen>)?)
- (OBJECT (<lparen> <unsigned integer> <rparen>)?)
- (BLOB (<lparen> <unsigned integer> <rparen>)?)
- (CLOB (<lparen> <unsigned integer> <rparen>)?)
- JSON

- (VARBINARY (<lparen> <unsigned integer> <rparen>)?)
- GEOMETRY
- GEOGRAPHY
- XML

A non-collection data type.

Example:

```
STRING
```

basic data type ::=

- <simple data type> (<lsbrace> <rsbrace>)*

A data type.

Example:

```
STRING[]
```

data type ::=

- <basic data type>
- ((<identifier> | <basicNonReserved>) (<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>
- <create role>
- <drop role>
- <GRANT>
- <Revoke GRANT>
- <CREATE POLICY>
- <DROP POLICY>
- <drop server>
- <drop table>
- <Import foreign schema>
- <Import another Database>
- <create database>
- <use database>
- <drop schema>
- <set schema>
- <create schema>
- <create procedure> (<ddl statement>)?
- <create data wrapper>
- <Drop data wrapper>
- <drop procedure>

A data definition statement.

Example:

```
CREATE FOREIGN TABLE X (Y STRING)
```

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 OPTIONS(x 'y')
```

use database ::=

- USE DATABASE <identifier> (VERSION <string>)?

database into working context

Example:

```
USE DATABASE foo
```

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);
```

drop schema ::=

- DROP (VIRTUAL | FOREIGN)? SCHEMA <identifier>

drop a schema in database

Example:

```
DROP SCHEMA foo
```

set schema ::=

- SET SCHEMA <identifier>

set the schema for subsequent ddl statements

Example:

```
SET SCHEMA foo
```

create a domain or type alias ::=

- CREATE DOMAIN (<identifier> | <basicNonReserved>) (AS)? <data type> (NOT NULL)?

creates a named type with optional constraints

Example:

```
CREATE DOMAIN my_type AS INTEGER NOT NULL
```

create data wrapper ::=

- CREATE FOREIGN ((DATA WRAPPER) | TRANSLATOR) <Unqualified identifier> ((TYPE | HANDLER) <identifier>)? (<options clause>)?

Defines a translator; use the options to override the translator properties.

Example:

```
CREATE FOREIGN DATA WRAPPER wrapper OPTIONS (x true)
```

Drop data wrapper ::=

- DROP FOREIGN ((DATA WRAPPER) | TRANSLATOR) <identifier>

Deletes a translator

Example:

```
DROP FOREIGN DATA WRAPPER wrapper
```

create role ::=

- CREATE ROLE <Unqualified identifier> (WITH <with role>)?

Defines data role for the database

Example:

```
CREATE ROLE lowly WITH FOREIGN ROLE "role"
```

with role ::=

- (ANY AUTHENTICATED)
- ((JAAS | FOREIGN) ROLE <identifier list>)

drop role ::=

- DROP ROLE <identifier>

Removes data role for the database

Example:

```
DROP ROLE <data-role>
```

CREATE POLICY ::=

- CREATE POLICY <identifier> ON ((<identifier> (FOR (ALL | ((SELECT | INSERT | UPDATE | DELETE) (<comma> (SELECT | INSERT | UPDATE | DELETE))*)))?) | (PROCEDURE <identifier> (FOR ALL)?)) TO <identifier> USING <lparen> <boolean primary> <rparen>

CREATE row level policy

Example:

```
CREATE POLICY pname ON tbl FOR SELECT,INSERT TO role USING col = user();
```

DROP POLICY ::=

- DROP POLICY <identifier> ON (<identifier> | (PROCEDURE <identifier>)) TO <identifier>

DROP row level policy

Example:

```
DROP POLICY pname ON tbl TO role
```

GRANT ::=

- GRANT (((<grant type> (<comma> <grant type>)*)? ON ((TABLE <identifier> (CONDITION ((NOT)? CONSTRAINT)? <string>)?) | (FUNCTION <identifier>) | (PROCEDURE <identifier> (CONDITION ((NOT)? CONSTRAINT)? <string>)?) | (SCHEMA <identifier>) | (COLUMN <identifier> (MASK (ORDER <unsigned integer>)? <string>)? (CONDITION (<boolean primary> | <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 ::=

- CREATE SERVER <Unqualified identifier> (TYPE <string>)? (VERSION <string>)? FOREIGN ((DATA WRAPPER) | TRANSLATOR) <Unqualified identifier> (<options clause>)?

Defines a connection to a source

Example:

```
CREATE SERVER "h2-connector" FOREIGN DATA WRAPPER h2 OPTIONS ("resource-name" 'java:/accounts-ds');
```

---

drop server ::=

- 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 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 VIEW name
```

create foreign temp table ::=

- CREATE (LOCAL)? FOREIGN TEMPORARY TABLE <Unqualified identifier> <create table body> ON <identifier>

Defines a foreign temp table

Example:

```
CREATE FOREIGN TEMPORARY TABLE t (x string) ON z
```

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 COLUMN x xml
```

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 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 vw INSTEAD OF INSERT ENABLED
```

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 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 FROM SERVER bar
```

Import another Database ::=

- IMPORT DATABASE <identifier> VERSION <string> (WITH ACCESS CONTROL)?

imports another database into current database

Example:

```
IMPORT DATABASE vdb VERSION '1.2.3' WITH ACCESS CONTROL]
```

identifier list ::=

- <identifier> (<comma> <identifier>)*

grant type ::=

- SELECT

- INSERT
- UPDATE
- DELETE
- EXECUTE
- ALTER
- DROP

Security Guide

The Teiid system provides a range of built-in and extensible security features to enable secure data access. This introduction provides a high-level guide to security concerns. The rest of the guide provides specifics on configuring clients, the Teiid server, and the application server.

Elytron Configuration

Examples in this guide are based upon the legacy security framework - which is still supported. If you want to migrate to the newer Elytron system you should remove the legacy teiid-security security domain and instead work with an analogous one created in Elytron. This also requires telling undertow to use the new security domain.

The standalone cli for this change to Elytron:

```
/subsystem=security/security-domain=teiid-security:remove()
/subsystem=elytron/security-domain=teiid-security:add(realms=[{realm=ApplicationRealm}])
/subsystem=undertow/application-security-domain=teiid-security:add(security-domain=teiid-security)
reload
```

From there you would not completely follow the examples shown here, but rather use them as a guide to follow migration documentation such as Migrate Legacy Security.

Authentication

Client Authentication

JDBC/ODBC/Web Service clients may use simple passwords to authenticate a user.

Typically a user name is required, however user names may be considered optional if the identity of the user can be discerned by the password credential alone. In any case it is up to the configured security domain to determine whether a user can be authenticated. If you need authentication, the administrator must configure LoginModules for Teiid.

Caution	By default, access to Teiid is NOT secure. The default LoginModules are only backed by file based authentication, which has a well known user name and password. We **DO NOT** recommend leaving the default security profile as defined when you are exposing sensitive data.

Teiid JDBC/ODBC also supports Kerberos authentication with additional configuration.

Auto-generated web services, such as OData, for consuming Teiid typically support HTTPBasic authentication, which in turn should utilize Pass-through Authentication.

Source Authentication

Source authentication is generally determined by the capabilities of JCA resource adapters used to connect to external resources. Consult the AS JCA documentation for the capabilities of source pooling and supplied resource adapters for more information. Typically a single username/password credential is supported, such as when creating JDBC Data Sources. In more advanced usage scenarios the source and/or translator may be configured or customized to use an execution payload, the Teiid subject, or even the calling application subject via Pass-through Authentication. See also Developing JEE Connectors and Translator Development

Pass-through Authentication

If your client application (web application or Web service) resides in the same WildFly instance as Teiid and the client application uses a security domain, then you can configure Teiid to use the same security domain and not force the user to re-authenticate. In pass-through mode Teiid looks for an authenticated subject in the calling thread context and uses it for sessioning and authorization. To configure Teiid for pass-through authentication, change the Teiid security-domain name to the same name as your application's security domain name. This change can be made via the CLI or in the `standalone-teiid.xml` file if running in standalone mode. The security domain must be a JAAS based LoginModule and your client application MUST obtain its Teiid connection using a Local Connection with the _PassthroughAuthentication=true connection flag set. You may also set the security-domain on the VDB.

Authorization

Authorization covers both administrative activities and data roles. A data role is a collection of permissions (also referred to as entitlements) and a collection of entitled principals or groups. With the deployment of a VDB the deployer can choose which principals and groups have which data roles. Check out Reference Guide Data Roles chapter for more information. Any source level authorization decisions are up to the source systems being integrated.

VDBs without data roles defined are accessible 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.

Encryption

Teiid Transports

Teiid provides built-in support for JDBC/ODBC over SSL. JDBC defaults to just sensitive message encryption (login mode), while ODBC (the pg transport) defaults to just clear text passwords if using simple username/password authentication.

The AS instance must be configured for SSL as well so that Any web services consuming Teiid may use SSL.

Configuration

Passwords in configuration files are by default stored in plain text. If you need these values to be encrypted, please see encrypting passwords for instructions on encryption facilities provided by the container.

Source Access

Encrypting remote source access is the responsibility for the resource adapter and library/driver used to access the source system.

Temporary Data

Teiid temporary data which can be stored on the file system as configured by the BufferManager may optionally be encrypted. Set the `buffer-service-encrypt-files` property to true on the Teiid subsystem to use 128-bit AES to encrypt any files written by the BufferManager. A new symmetric key will be generated for each start of the Teiid system on each server. A performance hit will be seen for processing that is memory intensive such that data typically spills to disk. This setting does not affect how VDBs (either the artifact or an exploded form) or log files are written to disk.

LoginModules

LoginModules are an essential part of the JAAS security framework and provide Teiid customizable user authentication and the ability to reuse existing LoginModules defined for WildFly. Refer to the WildFly security documentation for information about configuring security in WildFly, http://docs.jboss.org/jbossas/admindevel326/html/ch8.chapter.html.

Teiid can be configured with multiple named application policies that group together relevant LoginModules. These security-domain names can be referenced on a per vdb.

The security-domain attribute under the authentication element in `teiid` subsystem in the `<jboss-install>/standalone/configuration/standalone-teiid.xml` file is used set the security-domain name. For example, in default configuration under `teiid` subsystem you will find

```
<authentication security-domain="teiid-security"/>

<transport name="jdbc" protocol="teiid" socket-binding="teiid-jdbc">
   <ssl mode="login"/>
</transport>
```

If no domain can authenticate the user, the login attempt will fail. Details of the failed attempt including invalid users, which domains were consulted, etc. will be in the server log with appropriate levels of severity.

security-domain in VDB

A VDB can be configured to use a security-domain other than the Teiid default security-domain. This configuration is defined in the vdb.xml file, see VDB Properties for more information. The security-domain defined on transport configuration will be used as default security-domain, if a security-domain is not configured for a specific VDB.

```
<vdb name="vdb" version="1">
    <property name="security-domain" value="custom-security" />
    ...
</vdb>
```

Tip	In existing installations an appropriate security domain may already be configured for use by administrative clients (typically for `admin-console`). If the admin connections (CLI) are not secured, it is recommended that you secure that interface by executing `add-user.sh` script in the `bin/scripts` directory.

Built-in LoginModules

JBossAS provides several LoginModules for common authentication needs, such as authenticating from a Text Based LoginModule or a LDAP Based LoginModule.

You can install multiple login modules as part of single security domain configuration and configure them to be part of the login process. For example, for `teiid-security` domain, you can configure a file based and also LDAP based login modules, and have your user authenticated with either or both login modules. If you want to write your own custom login module, refer to the Developer’s Guide for instructions.

For all the available login modules refer to http://community.jboss.org/docs/DOC-11287.

Realm Based LoginModule

The *RealmDirectLoginModule* utilizes a common security realm across installed WildFly/EAP instance defined by default ApplicationRealm to perform authentication and authorization. To use this security relam add the following XML under "security" subsystem in standalone-teiid.xml or domain.xml

standalone-teiid.xml

```
 <subsystem xmlns="urn:jboss:domain:security:1.1">
    <security-domains>
        <security-domain name="teiid-security" cache-type="default">
            <authentication>
                <login-module code="RealmDirect" flag="required">
                    <module-option name="password-stacking" value="useFirstPass"/>
                </login-module>
            </authentication>
        </security-domain>
    </security-domains>
</subsystem>
```

When using this security domain, use *<wildfly>/bin/add-user.sh* or *<wildfly>/bin/add-user.bat* scripts to add/update a user in "ApplicationRelam". When using this relam, the password as stored in encrypted form. This is the default security module that is used.

Text Based LoginModule

The *UsersRolesLoginModule* utilizes simple text files to authenticate users and to define their groups. To use this add the following XML under "security" subsystem in standalone-teiid.xml or domain.xml

standalone-teiid.xml

```
 <subsystem xmlns="urn:jboss:domain:security:1.1">
    <security-domains>
        <security-domain name="teiid-security" cache-type="default">
            <authentication>
                <login-module code="UsersRoles" flag="required">
                    <module-option name="usersProperties" value="${jboss.server.config.dir}/users.properties"/>
                    <module-option name="rolesProperties" value="${jboss.server.config.dir}/roles.properties"/>
                </login-module>
            </authentication>
        </security-domain>
    </security-domains>
</subsystem>
```

Warning	The *UsersRolesLoginModule* is not recommended for production use and is strongly recommended that you replace this login module.

Per above configuration, User names and passwords are stored in the *<wildfly>/standalone/configuration/users.properties* file, an example user.properties file looks like below

users.properties

```
# A users.properties file for use with the UsersRolesLoginModule
# username=password

fred=password
george=password
...
```

The role assignments are stored in the *<wildfly>/standalone/configuration/roles.properties* file, an example roles.properties file looks like below

roles.properties

```
# A roles.properties file for use with the UsersRolesLoginModule
# username=role1,role2,...

data_role_1=fred,sally
data_role_2=george
```

User and role names are entirely up to the needs of the given deployment. For example each application team can set their own security constraints for their VDBs, by mapping their VDB data roles to application specific JAAS roles, e.g. app_role_1=user1,user2,user3.

Note	When you configure this security domain, you must provide the empty user.properties and roles.properties files at the correct path defined in the configuration, otherwise the initialization of security domain will endup in failure.

Note	Teiid data roles names are independent of JAAS roles. VDB creators can choose whatever name they want for their data roles, which are then mapped at deployment time to JAAS roles.

LDAP Based LoginModule

For more complete information to configure a LDAP based login module consult EAP documentation

Configure LDAP authentication by editing *standalone-teiid.xml* under 'security' subsystem. Once the security-domain is defined, then edit the 'security-domain' attribute for Teiid's 'transport' for which you want use this LDAP login.

standalone-teiid.xml

```
<subsystem xmlns="urn:jboss:domain:security:1.1">
    <security-domains>
        <security-domain name="ldap_security_domain">
            <authentication>
                <login-module code="LdapExtended" flag="required">
                    <module-option name="java.naming.factory.initial" value="com.sun.jndi.ldap.LdapCtxFactory"
/>
                    <module-option name="java.naming.provider.url" value="ldap://mydomain.org:389" />
                    <module-option name="java.naming.security.authentication" value="simple" />
                    <module-option name="bindDN" value="myuser" />
                    <module-option name="bindCredential" value="mypasswd" />
                    <module-option name="baseCtxDN" value="ou=People,dc=XXXX,dc=ca" />
                    <module-option name="baseFilter" value="(cn={0})" />
                    <module-option name="rolesCtxDN" value="ou=Webapp-Roles,ou=Groups,dc=XXXX,dc=ca" />
                    <module-option name="roleFilter" value="(member={1})" />
                    <module-option name="uidAttributeID" value="member" />
                    <module-option name="roleAttributeID" value="cn" />
                    <module-option name="roleAttributeIsDN" value="true" />
                    <module-option name="roleNameAttributeID" value="cn" />
                    <module-option name="roleRecursion" value="-1" />
                    <module-option name="searchScope" value="ONELEVEL_SCOPE" />
                    <module-option name="allowEmptyPasswords" value="false" />
                    <module-option name="throwValidateError" value="true" />
                </login-module>
            </authentication>
        </security-domain>
    </security-domains>
</subsystem>
```

Note	If using SSL to the LDAP server, ensure that the Corporate CA Certificate is added to the JRE trust store.

Note	Sometimes role information is DN, then you will requirethe property "parseRoleNameFromDN=true".

Database LoginModule

For information to configure a Database based login module consult EAP documentation

Cert LoginModule

For more complete information to configure a Certificate based login module consult EAP documentation

Role Mapping LoginModule

If the LoginModule you are using exposes role names that you wish to map to more application specific names, then you can use the RoleMappingLoginModule. This uses a properties file to inject additional role names, and optionally replace the existing role, on authenticated subjects.

standalone-teiid.xml

```
<subsystem xmlns="urn:jboss:domain:security:1.1">
    <security-domains>
        <security-domain name="ldap_security_domain">
            <authentication>
                ...
                <login-module code="org.jboss.security.auth.spi.RoleMappingLoginModule" flag="optional">
                    <module-option name="rolesProperties" value="${jboss-install}/standalone/configuration/role
s.properties" />
                    <module-option name="replaceRole" value="false" />
                </login-module>
                ...
            </authentication>
        </security-domain>
    </security-domains>
</subsystem>
```

Custom LoginModules

If your authentication needs go beyond the provided LoginModules, please refer to the JAAS development guide at http://java.sun.com/j2se/1.5.0/docs/guide/security/jaas/JAASLMDevGuide.html. There are also numerous guides available.

If you are extending one of the built-in LoginModules, refer to http://community.jboss.org/docs/DOC-9466.

Teiid Server Transport Security

There are two types of direct remote transports, each with it’s own encryption configuration:

- "teiid" - Defaults to only encrypt login traffic, in which none of the other configuration properties are used.
- "odbc" - Defaults to no SSL

Example Transport Configuration

```
<authentication security-domain="teiid-security"/>
<transport name="jdbc" socket-binding="teiid-jdbc" protocol="teiid"/>
<transport name="odbc" socket-binding="teiid-odbc" protocol="pg">
    <ssl mode="disabled"/>
</transport>
```

Warning	The pg protocol for ODBC access defaults to clear text username password authentication. You should consider using a security domain that utilizes non-plaintext passwords, kerberos, or SSL.

SSL configuration is part of the *transport* configuration in the Teiid subsystem.

Other indirect access into Teiid, such as OData or REST via WARs, relies on the container settings for HTTP/HTTPS access.

Encryption Modes

Teiid supports a couple different encryption modes based on the *mode* attribute on *ssl* element.

- **logIn** - This is the default setting for the transports.
  - JDBC (non-data by default) messages between client and server are encrypted using 128 bit AES with a Diffie-Hellman key that is negotiated per connection. When possible a 2048 bit key exchange will be used otherwise 1024 bit will be used. Oracle/Sun 1.7 JREs are known not to support key lengths over 1024 bits. The connection property encryptRequest can be used to encrypt requests and results using the same 128 bit AES scheme.
  - pg authentication is expected to be secure - which currently is only GSS logins. Pre 9.x and unpatched client/server combinations will use a less secure ECB block mode, which is not recommended for large authentication payloads and the encryptRequest option.
- **enabled** - Mode to enable SSL. Clients are required to connect using SSL.
- **disabled** - turns off any kind of encryption. This is the default for the odbc transport.

SSL Client Authentication Modes

- **anonymous** – **DEPRECATED** No certificates are required, but all communications are still encrypted using the TLS_DH_anon_WITH_AES_128_CBC_SHA SSL cipher suite. In most secure intranet environments anonymous is suitable to just bulk encrypt traffic without the need to setup SSL certificates. No certificates are exchanged, and settings are not needed for the keystore and truststore properties. JDBC Clients must have 'org.teiid.ssl.allowAnon' set to true (the default) to connect to an anonymous server.

Note	ODBC clients and some VMs, such as IBM, may not have the TLS_DH_anon_WITH_AES_128_CBC_SHA cipher suite available. On other VMs it may require modifying the java.security file to enable the anon cipher suites.

When the client or server lack the anonymous cipher suite, consider using server only or 1-way authentication with a self-signed certificate. ODBC clients typically do not require server certificate validation. Teiid JDBC clients by default validate the server certificate, but can use the org.teiid.ssl.trustAll property to accept any server certificate.

This mode is deprecated, as you may set the authentication mode to 1-way and the enabled-cipher-suites to TLS_DH_anon_WITH_AES_128_CBC_SHA for the same effect.

- **NONE**, previously known as **1-way**, is the default. Only authenticates the server to the client. Requires a private key keystore to be created for the server. If the client is configured to validate the server certificate, the client will need an appropriate truststore configured.
- **NEED**, previously known as **2-way**, is mutual client and server authentication. The server and client applications each have a keystore for their private keys and each has a truststore that authenticates the other. The server will present a certificate, which is obtained from the keystore related properties. The client should have a truststore configured to accept the server certificate. The client is also required to present a certificate, which is obtained from its keystore. The client certificate should be accepted by the trust store configured by the truststore related properties.
- **WANT** – Similar to NEED, but the client is not required to authenticate.

For non-anonymous SSL, the suite is negotiated - see *enabled-cipher-suites* below below.

Depending upon the SSL mode, follow the guidelines of your organization around creating/obtaining private keys. If you have no organizational requirements, then follow this guide to create self-signed certificates with their respective keystores and truststores. The following keystore and truststore combinations are required for different SSL modes. The names of the files can be chosen by the user. The following files are shown for example purposes only.

For any non-anonymous server configuration, you must supply

1. server.keystore - has server’s private key
2. server.truststore - has server’s public key

For NEED or WANT client authentication, a client may also need to provide

1. client.keystore - client’s private key
2. client.truststore - has client’s public key

Full Configuration Options

Example XML Configuration

```
<ssl mode="enabled" authentication-mode="NONE" ssl-protocol="TSLv1" keymanagement-algorithm="algo"
         enabled-cipher-suites="SSL_RSA_WITH_RC4_128_MD5,SSL_RSA_WITH_RC4_128_SHA">
            <keystore name="cert.keystore" password="passwd" type="JKS" key-alias="alias" key-password="passwd1"
/>
            <truststore name="cert.truststore" password="passwd"/>
</ssl>
```

Properties

- mode - diabled|login|enabled *disabled* = no transport or message level security will be used. *login* = only the login traffic will be encrypted at a message level using 128 bit AES with an ephemeral DH key exchange. Only applies to the `teiid` transport and no other config values are needed in this mode. *enabled* = traffic will be secured with SSL using the other configuration properties. `teiid` transport clients **must** connect using SSL with the mms protocol. ODBC "pg" transport clients may optionally use SSL.
- ssl-protocol- Type of SSL protocol to be used. Optional - by default TLSv1.

Caution	SSLv3 is not recommended due to the POODLE security vulnerability.

- keystore/type - Keystore type created by the keytool. Optional - by default "JKS" is used.
- authentication-mode - NONE|NEED|WANT - Type of SSL Client Authentication Mode.
- keymanagement-algorithm - Type of key algorithm used. Optional - by default is based upon the VM, e.g. "SunX509"
- keystore/name - The file name of the keystore, which contains the private key of the Server. The file name can be relative resource path available to the Teiid deployer classloader or an absolute file system path. A typical installation would place the keystore file in the conf directory of the profile where Teiid is deployed with a file name relative to the conf path. Typically required for non-anonymous authentication.
- keystore/password - password for the keystore. Required if the keystore has a password.
- keystore/key-alias - Alias name for the private key to use. Optional - only needed if there are multiple private keys in the keystore and you need to choose which one to use.
- keystore/key-password - Alias name for the private key to use. Optional - only needed if the key password is different than the keystore password.
- truststore/name - This is the truststore containing the public certificate(s) for client keys. Depending upon how you created the keystore and truststores, this may be same file as defined under "keystore/name" property. Required if "authenticationMode" is "WANT" or "NEED".
- truststore/password - password for the truststore. Required if the truststore has a password.
- truststore/check-expired - Whether to check for expired client certificates. Default false.
- enabled-cipher-suites - A comma separated list of cipher suites allowed for encryption between server and client. The values must be valid supported cipher suites otherwise SSL connections will fail. Optional - defaults to all supported cipher suites for the vm.

Alternatively, you can use the CLI to add or modify the transport configuration

```
/subsystem=teiid/transport=jdbc:write-attribute(name=ssl-mode,value=enabled)
/subsystem=teiid/transport=jdbc:write-attribute(name=ssl-authentication-
mode,value=1-way)
/subsystem=teiid/transport=jdbc:write-attribute(name=ssl-ssl-protocol,value=TLSv1)
/subsystem=teiid/transport=jdbc:write-attribute(name=ssl-keymanagement-
algorithm,value=SunX509)
/subsystem=teiid/transport=jdbc:write-attribute(name=ssl-enabled-cipher-
suites,value="SSL_RSA_WITH_RC4_128_MD5,SSL_RSA_WITH_RC4_128_SHA")
/subsystem=teiid/transport=jdbc:write-attribute(name=keystore-name,value=ssl-
example.keystore)
/subsystem=teiid/transport=jdbc:write-attribute(name=keystore-
password,value=redhat)
/subsystem=teiid/transport=jdbc:write-attribute(name=keystore-type,value=JKS)
/subsystem=teiid/transport=jdbc:write-attribute(name=keystore-key-
alias,value=teiid)
/subsystem=teiid/transport=jdbc:write-attribute(name=keystore-key-
password,value=redhat)
/subsystem=teiid/transport=jdbc:write-attribute(name=truststore-name,value=ssl-
example.truststore)
/subsystem=teiid/transport=jdbc:write-attribute(name=truststore-
password,value=redhat)
```

Note	If you do not like to leave clear text passwords in the configuration file, then you can use WildFly vault mechanism for storing the keystore and truststore passwords. Use the directions defined here https://community.jboss.org/docs/DOC-17248

Encryption Strength

Both anonymous SSL and login only (JDBC specific) encryption are configured to use 128 bit AES encryption by default. By default non-anonymous SSL allow for cipher suite negotiation based upon the default cipher suites supported by the respective Java platforms of the client and server. Users can restrict the cipher suites used by specifying the *enabled-cipher-suites* property above in the SSL configuration.

Examples

- 1-way ssl authentication mode

JDBC/ODBC SSL connection using self-signed SSL certificates

When you are operating in a secure environment, you need to think about mutual authentication with the server you connecting to and also encrypt all the messages going back and forth between the client and server. In Teiid, both JDBC and ODBC protocols support SSL based connections. Typically for development purposes you will not have CA signed certificates, and you need to validate with self-signed certificates. In article, I will show the steps to generate a self-signed certificate and then configuring them in Teiid. Then configuring the JDBC and ODBC clients with the defined SSL certificates to communicate with the Teiid server.

Creating self-signed certificates

If you do not already have it, download the "openssl" libraries for your environment. Follow the below script for creating the certificate(s).

Create root CA Certificate

To begin with, you need to generate the root CA key (this is what signs all issued certs), make sure you give a strong pass phrase.

```
openssl genrsa -des3 -passout pass:changeme  -out rootCA.key 2048
openssl rsa -passin pass:changeme -in rootCA.key -out rootCA.key
```

Generate the self-signed (with the key previously generated) root CA certificate:

```
openssl req -new -key rootCA.key -out rootCA.csr
openssl req -x509 -in rootCA.csr -key rootCA.key -days 365 -out rootCA.crt
```

You can install this on Teiid Server machine that will be communicating with services using SSL certificates generated by this root certificate. Typically, you’ll want to install this on all of the servers on your internal network.

To work with Teiid server, you need to import this certificate into keystore. Follow the below steps

```
openssl pkcs12 -export -in rootCA.crt -inkey rootCA.key -out rootCA.p12 -noiter -
nomaciter -name root
keytool -importkeystore -destkeystore rootCA.keystore -srckeystore rootCA.p12 -
srcstoretype pkcs12 -alias root
```

Generating client side certificates

Once you have the root CA certificate generated, you can use that to generate additional SSL certificates for other JDBC or ODBC and for other services.

1-WAY SSL

For 1-WAY SSL, we would need to extract rootCA’s trust certificate (public key) and create a keystore with it.

```
openssl x509 -trustout -in rootCA.crt > rootCA_trust.crt
keytool -importcert -v -trustcacerts -alias rootCA -file rootCA_trust.crt -keystore
teiid.keystore
openssl x509 -in rootCA_trust.crt -out rootCA_trust.cer -outform der
```

Here we created keystore (teiid.keystore) that can be used with java based applications like JDBC driver, and also created certificate (rootCA_trust.cer) that can be used in Windows platform.

2-WAY SSL

for 2-WAY SSL, you would need an another certificate on client side. To create an SSL certificate you can use for one of your services, the first step is to create a certificate signing request (CSR). To do that, you need a key (separate from the root CA key you generated earlier). Then generate a CSR

```
openssl genrsa -out teiid.key 2048
openssl rsa -passin pass:changeme -in teiid.key -out teiid.key
```

Generate the self-signed certificate, and generate signed certificate using the root CA certificate and key you generated previously. Make sure the Common Name (CN) is set to the FQDN, hostname or IP address of the machine you’re going to put this on.

```
openssl req -new -key teiid.key -out teiid.csr
openssl x509 -req -in teiid.csr -CA rootCA.crt -CAkey rootCA.key -CAcreateserial -
out teiid.crt -days 365
```

Now you have an SSL certificate (in PEM format) called teiid.crt This is the certificate you want your JDBC or ODBC to use. Import this certificate into a existing key store or create a new one using

```
openssl pkcs12 -export -in teiid.crt -inkey teiid.key -out teiid.p12 -noiter -
nomaciter -name teiid
keytool -importkeystore -destkeystore teiid.keystore -srckeystore teiid.p12 -
srcstoretype pkcs12 -alias teiid
keytool -importcert -file rootCA_trust.crt -keystore teiid.keystore
```

Also, import the client certificate’s public key into rootCA keystore

```
openssl x509 -trustout -in teiid.crt > teiid_trust.crt
keytool -importcert -file teiid_trust.crt -keystore rootCA.keystore
```

I also found a great reference here [1] & [2] for certificate generation. Note in above that, I had issues with recognizing the PKCS12 formatted keystore in Java VM, I had to convert into a JKS format.

Configuring the Teiid Server with Certificates

- Install Teiid server if you do not already have one.
- Edit the standalone-teiid.xml file, and find "teiid" subsystem and inside find JDBC and ODBC transports and add as following.

```
<transport name="jdbc" socket-binding="teiid-jdbc" protocol="teiid">
```

```
    <ssl mode="enabled" authentication-mode="1-way">
        <keystore name="/path/to/rootCA.keystore" password="changeme" type="JKS"/>
      <!-- uncomment and configure for 2-way authentication
        <truststore name="/path/to/rootCA.keystore" password="changeme"/>
      -->
    </ssl>
</transport>
<transport name="odbc" socket-binding="teiid-odbc" protocol="pg">
    <ssl mode="enabled" authentication-mode="1-way">
        <keystore name="/path/to/rootCA.keystore" password="changeme" type="JKS"/>
      <!-- uncomment and configure for 2-way authentication
        <truststore name="/path/to/rootCA.keystore" password="changeme"/>
      -->
    </ssl>
</transport>
```

Then restart the server to start accepting the connections using SSL. Now server set up is complete.

Configuring JDBC client to use SSL

When using a JDBC client to use the SSL, copy the server.truststore file to the target machine. One of the main change is difference in JDBC connection URL you need to use. For example if your JDBC connection string is

```
jdbc:teiid:<vdb>:mm://<host>:31000
```

then change it to

```
jdbc:teiid:<vdb>:mms://<host>:31000
```

note "mm[s]" to represent [s] for secure. You also need to add the following system properties to your client for

1-WAY SSL

```
-Djavax.net.ssl.trustStore=/path/to/teiid.keystore
-Djavax.net.ssl.trustStorePassword=changeme
-Djavax.net.ssl.keyStoreType=JKS
```

2-WAY SSL

```
-Djavax.net.ssl.keyStore=/path/to/teiid.keystore
-Djavax.net.ssl.keyStorePassword=changeme
-Djavax.net.ssl.trustStore=/path/to/teiid.keystore
-Djavax.net.ssl.trustStorePassword=changeme
-Djavax.net.ssl.keyStoreType=JKS
```

The start your client application normally, that should make sure the SSL certificates used for encryption.

Configuring ODBC client to use SSL (Windows)

- Install the Postgresql ODBC driver in your Windows machine.

1-WAY SSL

- Copy the "*rootCA.crt*" and "*rootCA_trust.cer*" files into your Windows machine into directory *c:\Users\<yourname>\AppData\Roaming\postgresql*. Note this directory may be hidden or non existent, if non-existent create a new folder. Note that if you are dealing with CA signed certificate, you do not have to share your private certificate "rootCA.crt". However since we are using self signed this will become the root certificate.
- Rename "*rootCA.crt*" to "*root.crt*"
- Rename "*rootCA_trust.cer*" to "*postgresql.cer*"
- Now open the "ODBC Data Manager" application, create DSN for the connection you are ready to make using previously installed Postgres ODBC driver. Provide the correct host name and port (35432), and use VDB name as Database name, and select the "ssl-model" property to "verify-ca" or "verify-full" and save the configuration.

2-WAY SSL

- Copy the "*rootCA.crt*", "*teiid.crt*", "*teiid.key*" files into your Windows machine into directory *c:\Users\<yourname>\AppData\Roaming\postgresql*. Note this directory may be hidden or non existent, if non-existent create a new folder. Note that if you are dealing with CA signed certificate, you do not have to share your private certificate "rootCA.crt". However since we are using self signed this will become the root certificate.
- Rename "*rootCA.crt*" to "*root.crt*"
- Rename "*teiid.crt*" to "*postgresql.crt*"
- Rename "*teiid.key*" to "*postgresql.key*"
- Now open the "ODBC Data Manager" application, create DSN for the connection you are ready to make using previously installed Postgres ODBC driver. Provide the correct host name and port (35432), and use VDB name as Database name, and select the "ssl-model" property to "verify-ca" or "verify-full" and save the configuration.
- Now use any ODBC client application/tool like (QTODBC) and make ODBC connection using the DSN created and start issuing the SQL queries.

Security at the Data Source Level

In some use cases, the user might need to pass-in different credentials to their data sources based on the logged in user rather than using the shared credentials for all the logged users. To support this feature, WildFly and Teiid provide multiple login modules to be used in conjunction with Teiid's main security domain. See this document for details on configuration. Note that these directions need to be used in conjunction with the container document.

CallerIdentity

If client wants to pass in simple text password or a certificate or a custom serialized object as token credential to the data source, the admin can configure the "CallerIdentity" login module. Using this login module a user can pass-in their Teiid security domain login credential to the data source. Here is a sample configuration:

standalone-teiid.xml

```
<subsystem xmlns="urn:jboss:domain:security:1.1">
    <security-domains>
        <security-domain name="my-security-domain">
            <authentication>
                <login-module code="RealmDirect" flag="required">
                    <module-option name="password-stacking" value="useFirstPass"/>
                </login-module>

                <login-module code="org.picketbox.datasource.security.CallerIdentityLoginModule" flag="required"
>
                    <module-option name="password-stacking" value="useFirstPass"/>
                </login-module>
            </authentication>
        </security-domain>
    </security-domains>
</subsystem>
```

Note	This security domain should only be used to secure data sources, and not as generic purpose security domain.

Note	**"applicability"** - CallerIdentity Login module is only applicable when the logged in subject contains the text based credentials. The login module retrieves and uses the username and password for the data source authentication purposes. When working with non-character based passwords use Passthough Identity defined below.

In the datasource configuration, instead of supplying the username/password you need to add the following element:

In JDBC Datasource

```
<datasource jndi-name="java:/mysql-ds" pool-name="mysql-ds" enabled="true">
    <connection-url>jdbc:mysql://localhost:3306/txns</connection-url>
    <driver>mysql</driver>
     <pool>
          <allow-multiple-users>true</allow-multiple-users>
     </pool>
     <security>
          <security-domain>my-security-domain</security-domain>
     </security>
</datasource>
```

In a connection factory ex:ldap

```
          <resource-adapter>
```

```
            <archive>teiid-connector-ldap.rar</archive>
            <transaction-support>NoTransaction</transaction-support>
            <connection-definitions>
                <connection-definition class-name="org.teiid.resource.adapter.ldap.LDAPManagedConnectionFactory"

                        jndi-name="java:/ldapDS"
                        enabled="true"
                        use-java-context="true"
                        pool-name="ldap-ds">

                  <config-property name="LdapUrl">ldap://ldapServer:389</config-property>
                  <config-property name="LdapAdminUserDN">cn=???,ou=???,dc=???</config-property>
                  <config-property name="LdapAdminUserPassword">pass</config-property>
                  <config-property name="LdapTxnTimeoutInMillis">-1</config-property>

                  <security>
                     <security-domain>my-security-domain</security-domain>
                 </security>
                </connection-definition>
            </connection-definitions>
        </resource-adapter>
```

When user logs in with a password, the **same** username and password will be also set on the logged in Subject after authentication. These credentials can be extracted by the data source by asking for Subject's private credentials.

Please note that encoding and decoding of this credential is strictly up to the user as WildFly and Teiid will only act as a carrier of the information from login module to connection factory. Using this CallerIdentity module, the connection pool for data source is segmented by Subject.

Pass Through Identity

This is similar to the CallerIdentity login module, where the calling user's credentials and roles are passed as is. This is especially useful when dealing with non-text based credentials where you want to pass down the payload as is.

Note	this login module will typically only be used in OAuth delegation scenarios.

standalone-teiid.xml

```
<subsystem xmlns="urn:jboss:domain:security:1.1">
    <security-domains>
        <security-domain name="passthrough-security">
            <authentication>
                <login-module code="org.teiid.jboss.PassthroughIdentityLoginModule" flag="required" module="org
.jboss.teiid">
                    <module-option name="username" value="guest"/>
                    <module-option name="password" value="guest"/>
                </login-module>
            </authentication>
        </security-domain>
    </security-domains>
</subsystem>
```

Note	This security domain should only be used to secure data sources, and not as generic purpose security domain.

In the datasource configuration, instead of supplying the username/password you need to add the following element

In JDBC Datasource

```
<datasource jndi-name="java:/mysql-ds" pool-name="mysql-ds" enabled="true">
    <connection-url>jdbc:mysql://localhost:3306/txns</connection-url>
    <driver>mysql</driver>
```

```
    <pool>
          <allow-multiple-users>true</allow-multiple-users>
    </pool>
    <security>
          <security-domain>passthrough-security</security-domain>
    </security>
</datasource>
```

OAuth Authentication

Secured Rest services with OAuth authentication can be used in Teiid, however the data sources need to be configured with OAuth Refresh Token or Json Web Token (JWT) based security domains.

Refresh Token

A connected application is different among vendors like Google, LinkedIn, SalesForce etc. For details about creating a connected application consult the vendor's documentation. Once you have created a connected application, then run *teiid-oauth-util.sh* in "<eap>/bin" directory, use client_id, client_pass, and call back from source specific connected application. This script will provide the necessary values to plug-in below CLI script.

create a security-domain by executing CLI

```
/subsystem=security/security-domain=oauth2-security:add(cache-type=default)
/subsystem=security/security-domain=oauth2-security/authentication=classic:add
/subsystem=security/security-domain=oauth2-security/authentication=classic/login-module=oauth:add(code=org.teii
d.jboss.oauth.OAuth20LoginModule, flag=required, module=org.jboss.teiid.security,
   module-options=[client-id=xxxx, client-secret=xxxx, refresh-token=xxxx,
   access-token-uri=https://login.salesforce.com/services/oauth2/token])
reload
```

this will generate the following XML in the standalone.xml or domain.xml (this can also be directly added to the standalone.xml or domain.xml files instead of executing the CLI)

standalone.xml

```
<security-domain name="oauth2-security">
    <authentication>
        <login-module code="org.teiid.jboss.oauth.OAuth20LoginModule" flag="required" module="org.jboss.teiid.s
ecurity">
            <module-option name="client-id" value="xxxx"/>
            <module-option name="client-secret" value="xxxx"/>
            <module-option name="refresh-token" value="xxxx"/>
            <module-option name="access-token-uri" value="https://login.salesforce.com/services/oauth2/token"/>
        </login-module>
    </authentication>
</security-domain>
```

JSON Web Token (JWT)

A connected application is different among vendors like Google, LinkedIn, SalesForce etc. For details about creating a connected application consult the vendor's documentation. Once you have created connected application that uses the JWT, gather the below information client-id, client-secret, access-token-uri, jwt-audience,jwt-subject,keystore-type,keystore-password, keystore-url,certificate-alias,signature-algorithm-name and provide in the below CLI. (only tested with SalesForce)

```
/subsystem=security/security-domain=oauth2-jwt-security:add(cache-type=default)
/subsystem=security/security-domain=oauth2-jwt-security/authentication=classic:add
/subsystem=security/security-domain=oauth2-jwt-security/authentication=classic/login-module=oauth:add(code=org.
teiid.jboss.oauth.OAuth20LoginModule, flag=required, module=org.jboss.teiid.security,
```

```
    module-options=[client-id=xxxx, client-secret=xxxx, access-token-uri=https://login.salesforce.com/services/o
auth2/token, jwt-audience=https://login.salesforce.com, jwt-subject=your@sf-login.com,
    keystore-type=JKS, keystore-password=changeme, keystore-url=${jboss.server.config.dir}/salesforce.jks, cert
ificate-alias=teiidtest, signature-algorithm-name=SHA256withRSA])
reload
```

this will generate following XML in the standalone.xml or domain.xml (this can also be directly added to the standalone.xml or domain.xml files instead of executing the CLI)

standalone.xml

```
<security-domain name="oauth2-jwt-security">
    <authentication>
        <login-module code="org.teiid.jboss.oauth.JWTBearerTokenLoginModule" flag="required" module="org.jboss.
teiid.security">
            <module-option name="client-id" value="xxxxx"/>
            <module-option name="client-secret" value="xxxx"/>
            <module-option name="access-token-uri" value="https://login.salesforce.com/services/oauth2/token"/>
            <module-option name="jwt-audience" value="https://login.salesforce.com"/>
            <module-option name="jwt-subject" value="your@sf-login.com"/>

            <module-option name="keystore-type" value="JKS"/>
            <module-option name="keystore-password" value="changeme"/>
            <module-option name="keystore-url" value="${jboss.server.config.dir}/salesforce.jks"/>
            <module-option name="certificate-alias" value="teiidtest"/>
            <module-option name="signature-algorithm-name" value="SHA256withRSA"/>
        </login-module>
    </authentication>
</security-domain>
```

Kerberos

Kerberos can also used as data source security. The below configuration is to configure a static Kerberos ticket at data source. Please note that Kerberos can be used with RDBMS, REST web services.

```
/subsystem=security/security-domain=host:add(cache-type=default)
/subsystem=security/security-domain=host/authentication=classic:add
/subsystem=security/security-domain=host/authentication=classic/login-module=Kerberos:add(code=Kerberos, flag=r
equired,
   module-options=[storeKey=true, refreshKrb5Config=true, useKeyTab=true,
   principal=host/testserver@MY_REALM, keyTab=/path/to/service.keytab, doNotPrompt=true, debug=false])
reload
```

The above command will generate resulting XML in the standalone.xml file or domain.xml file.

standalone.xml

```
<security-domain name="host">
   <authentication>
      <login-module code="Kerberos" flag="required">
         <module-option name="storeKey" value="true"/>
         <module-option name="useKeyTab" value="true"/>
         <module-option name="principal" value="host/testserver@MY_REALM"/>
         <module-option name="keyTab" value="/path/to/service.keytab"/>
         <module-option name="doNotPrompt" value="true"/>
         <module-option name="debug" value="false"/>
         <module-option name="refreshKrb5Config" value = "true"/>
         <module-option name="addGSSCredential" value = "true"/>
      </login-module>
   </authentication>
</security-domain>
```

Kerberos Delegation

For using the same kerberos token at Teiid and as well as at the data source level, the token negotiated at the Teiid engine can be passed into data source. The data source must be configured to support this. Major database vendors like Oracle, MS-SQLServer, DB2, HIVE, Impala support kerberos. Some also support pass through mode. To make delegation work, follow the directions here to setup the Kerberos at Teiid engine level [Kerberos support through GSSAPI] and use the module option delegationCredential:

```
<module-option name="delegationCredential" value="USE"/>
```

Tip	When working with Kerberos/GSS security token (GssCredential), some JDBC drivers (MS-SQLServer) upon close of the connection they invalidate the GssCredential security token, to avoid accidental invalidation, add an option to above security-domain’s login-module configuration to wrap the passed in security token by adding below configuration

```
<module-option name="wrapGSSCredential" value="true"/>
```

Translator Customization

Teiid’s extensible Translator framework also provides hooks for securing access at the DataSource level. The `ExecutionFactory.getConnection` may be overridden to initialize the source connection in any number of ways, such as re-authentication, based upon the Teiid `Subject` , execution payload, session variables, and any of the other relevant information accessible via the `ExecutionContext` and the `CommandContext` . You may even also modify the generated source SQL in any way that is seen fit in the relevant `Execution` .

Kerberos support through GSSAPI

Teiid supports kerberos authentication using GSSAPI for single sign-on applications. This service ticket negotiation based authentication is supported through remote JDBC/ODBC drivers and LocalConnections. Client configuration is different for each client type.

LocalConnection

Set the JDBC URL property *PassthroughAuthentication* as true and use JBoss Negotiation for authentication of your web-application with kerberos. When the web application authenticates with the provided kerberos token, the same subject authenticated will be used in Teiid. For details about configuration, check the configuring the SSO with Kerberos in EAP

Server configuration for Remote JDBC/ODBC Connections

To support kerberos SSO on remote JDBC and ODBC connections, both client side and server side configurations need to be modified. On the server side, EAP needs to be configured with two different login modules. The below CLI script shows examples of it. Make necessary changes related to your configuration in terms of key tab locations, service principal etc.

Configure security domain to represent the identity of the server.

The first security domain authenticates the container itself to the directory service. It needs to use a login module which accepts some type of static login mechanism, because a real user is not involved. This example uses a static principal and references a keytab file which contains the credential.

```
/subsystem=security/security-domain=host:add(cache-type=default)
/subsystem=security/security-domain=host/authentication=classic:add
/subsystem=security/security-domain=host/authentication=classic/login-module=Kerberos:add(code=Kerberos, flag=r
equired,
module-options=[storeKey=true, refreshKrb5Config=true, useKeyTab=true,
principal=host/testserver@MY_REALM, keyTab=/path/to/service.keytab, doNotPrompt=true, debug=false])
reload
```

The above command will generate resulting XML in the standalone.xml file or domain.xml file.

standalone-teiid.xml

```
<security-domain name="host">
   <authentication>
      <login-module code="Kerberos" flag="required">
         <module-option name="storeKey" value="true"/>
         <module-option name="useKeyTab" value="true"/>
         <module-option name="principal" value="host/testserver@MY_REALM"/> <!-- service principal -->
         <module-option name="keyTab" value="/path/to/service.keytab"/>
         <module-option name="doNotPrompt" value="true"/>
         <module-option name="debug" value="false"/>
         <module-option name="refreshKrb5Config" value = "true"/>
      </login-module>
   </authentication>
</security-domain>
```

Configure security domain to secure the Teiid application.

The second security domain is used to authenticate the individual user to the Kerberos server. You need at least one login module to authenticate the user, and another to search for the roles to apply to the user. The following XML code shows an example SPNEGO security domain. It includes an authorization module to map roles to individual users. You can also use a module which searches for the roles on the authentication server itself. Note the name of security-domain MUST match realm. The following CLI script shows example of creating the login module

```
/subsystem=security/security-domain=MY_REALM:add(cache-type=default)
/subsystem=security/security-domain=MY_REALM/authentication=classic:add
/subsystem=security/security-domain=MY_REALM/authentication=classic/login-module=SPNEGO:add(code=SPNEGO, flag=r
equisite,
module-options=[serverSecurityDomain=host,password-stacking=useFirstPass])
/subsystem=security/security-domain=MY_REALM/authentication=classic/login-module=UserRoles:add(code=SPNEGO, fla
g=requisite,
module-options=[usersProperties=spnego-users.properties,rolesProperties=spnego-roles.properties])
reload
```

The above CLI will result in following result XML in standalone.xml or domain.xml depending upon configuration

standalone-teiid.xml

```
<security-domain name="MY_REALM">
   <authentication>
      <!-- Check the username and password -->
      <login-module code="SPNEGO"  flag="requisite">
         <module-option name="password-stacking" value="useFirstPass"/>
         <module-option name="serverSecurityDomain" value="host"/>
      </login-module>
      <!-- Search for roles -->
      <login-module code="UserRoles" flag="requisite">
         <module-option name="password-stacking" value="useFirstPass" />
         <module-option name="usersProperties" value="spnego-users.properties" />
         <module-option name="rolesProperties" value="spnego-roles.properties" />
      </login-module>
   </authentication>
</security-domain>
```

Note	**"User Roles/Groups associations"** Kerberos does not assign any user roles to the authenticated subject, that is reason you need to configure a separate role mapping module to assign roles. As an example in the above, "UserRoles" login-module is added. User need to edit ``"spnego-roles.properties" file and add groups in the format of `user@MY_REALM=my-group``. Check JBoss EAP documentation, as to all the available mapping modules that are available.

SPENGO security-domain delegates the calls relating to Kerberos to Kerberos server based on "serverSecurityDomain" property. If you would like configure the choice of authenticating using Kerberos or some other additional security domain on the same JDBC/ODBC transport, then you need to supply an additional module option (this can also be viewed as fallback authentication model)

```
<module-option name="usernamePasswordDomain" value="{user-name-based-auth}"/>
```

the resulting xml will look like below where {user-name-based-auth} replaced with a JAAS based simple username/password login module "app-fallback"

standalone-teiid.xml

```
<security-domain name="MY_REALM">
        <authentication>
                <!-- Check the username and password -->
                <login-module code="SPNEGO" flag="requisite">
                        <module-option name="password-stacking" value="useFirstPass"/>
                        <module-option name="serverSecurityDomain" value="host"/>
                        <module-option name="usernamePasswordDomain" value="app-fallback"/>
```

```
                </login-module>
                <!-- Search for roles -->
                <login-module code="UserRoles" flag="requisite">
                        <module-option name="password-stacking" value="useFirstPass" />
                        <module-option name="usersProperties" value="spnego-users.properties" />
                        <module-option name="rolesProperties" value="spnego-roles.properties" />
                </login-module>
        </authentication>
</security-domain>

<security-domain name="app-fallback" cache-type="default">
        <authentication>
                <login-module code="UsersRoles" flag="required">
                        <module-option name="usersProperties" value="file:${jboss.server.config.dir}/fallback-u
sers.properties"/>
                        <module-option name="rolesProperties" value="file:${jboss.server.config.dir}/fallback-r
oles.properties"/>
                </login-module>
        </authentication>
</security-domain>
```

Server Transport Configuration

The above configuration defined security-domains, before you can use these domains for login into Teiid, they need to be associated with Teiid’s transport configuration or VDB configuration. Paragraphs below offer both solutions.

Defining a "default" authentication

User can define a "default" authentication as below that can be used for all the VDBs system wide.

```
Use below CLI commands to edit the configuration
----
/subsystem=teiid:write-attribute(name=authentication-security-domain, value=MY_REALM)
/subsystem=teiid:write-attribute(name=authentication-type, value=GSS)
----
```

Will result in following changes (or you can edit the standalone-teiid.xml file directly)

```
<authentication security-domain="MY_REALM" type="GSS"/>
<transport name="jdbc" protocol="teiid" socket-binding="teiid-jdbc"/>
```

"What is the value of Type"

The "type" attribute above defines the type of authentication that needs to be enforced on the transport/vdb. The allowed values for type are

- USERPASSWORD - only allow user name/password based authentications
- GSS - only allow GSS API based authentication (Kerberos5).

Defining VDB based authentication

You can add following combination VDB properties in the vdb.xml file to select or force the security-domain and authentication type.

```
<property name="security-domain" value="MY_REALM" />
<property name="gss-pattern" value="{regex}" />
<property name="password-pattern" value="{regex}" />
<property name="authentication-type" value="GSS or USERPASSWORD" />
```

All the properties above are optional on a VDB. If you want to define VDB based security configuration "security-domain" property is required. If you want to enforce single authentication type use "authentication-type" property is required. If your security domain can support both GSS and USERPASSWORD, then you can define "gss-pattern" and "password-pattern" properties, and define a regular expression as the value. During the connection, these regular expressions are matched against the connecting user's name provided to select which authentication method user prefers. For example, if the configuration is defined as below

```
<property name="security-domain" value="MY_REALM" />
<property name="gss-pattern" value="logasgss" />
```

and if you passed the "user=logasgss" in the connection string, then GSS authentication is selected as login authentication mechanism. If the user name does not match, then default transport's authentication method is selected. Alternatively, if you want choose USERPASSWORD

```
<property name="security-domain" value="MY_REALM" />
<property name="password-pattern" value="*-simple" />
```

and if the user name is like "mike-simple", then that user will be subjected to authenticate against USERPASSWORD based authentication domain. You can configure different security-domains for different VDBS. VDB authentication will no longer be dependent upon underlying transport. If you like force "GSS" all the time then use configuration like below

```
<property name="security-domain" value="MY_REALM" />
<property name="authentication-type" value="GSS" />
```

Required System Properties on Server

JBoss EAP offers the ability to configure system properties related to connecting to Kerberos servers. Depending on the KDC, Kerberos Domain, and network configuration, the below system properties may or may not be required.

Edit the "standalone.conf" or domain.conf file in the "${jboss-as}/bin" directory and add the following JVM options \(changing the realm and KDC settings according to your environment)

```
JAVA_OPTS = "$JAVA_OPTS -Djava.security.krb5.realm=EXAMPLE.COM -
Djava.security.krb5.kdc=kerberos.example.com -
Djavax.security.auth.useSubjectCredsOnly=false"
```

or

```
JAVA_OPTS = "$JAVA_OPTS -Djava.security.krb5.conf=/path/to/krb5.conf -
Djava.security.krb5.debug=false -Djavax.security.auth.useSubjectCredsOnly=false"
```

or you can add these properties inside the standalone-teiid.xml file right after the <extensions> segment as

```
<system-properties>
    <property name="java.security.krb5.conf" value="/pth/to/krb5.conf"/>
    <property name="java.security.krb5.debug" value="false"/>
    <property name="javax.security.auth.useSubjectCredsOnly" value="false"/>
</system-properties>
```

This finishes the configuration on the server side, restart the server and make sure there are no errors during start up.

JDBC Client Configuration

Your workstation where the JDBC Client exists must have been authenticated using GSS API against Active Directory or Enterprise directory server. See this website http://spnego.sourceforge.net on instructions as to how to verify your system is authenticated into enterprise directory server. Contact your company’s operations team if you have any questions.

In your client VM the JAAS configuration for Kerberos authentication needs to be written. A sample configuration file (client.conf) is show below

"client.conf"

```
Teiid {
    com.sun.security.auth.module.Krb5LoginModule required
    useTicketCache=true
    storeKey=true
    useKeyTab=true
    keyTab="/path/to/krb5.keytab"
    doNotPrompt=true
    debug=false
    principal="user@EXAMPLE.COM";
};
```

Make sure you have configured the "keytab" properly, you can check this website for utilities and instructions to check your access to KDC server and to create keytab especially on windows environments http://spnego.sourceforge.net. For Redhat Linux see https://access.redhat.com/site/solutions/208173

Add the following JVM options to your client’s startup script - change Realm and KDC settings according to your environment

"Based on krb5.conf file"

```
-Djava.security.krb5.conf=/path/to/krb5.conf (default on Linux /etc/krb5.conf)
-Djava.security.auth.login.config=/path/to/client.conf
-Djavax.security.auth.useSubjectCredsOnly=false
-Dsun.security.krb5.debug=false
```

or

"Based on KDC and Realm file"

```
-Djava.security.krb5.realm=EXAMPLE.COM
-Djava.security.krb5.kdc=kerberos.example.com
-Djavax.security.auth.useSubjectCredsOnly=false
-Dsun.security.krb5.debug=false
-Djava.security.auth.login.config=/path/to/client.conf
```

Add the following additional URL connection properties to Teiid JDBC connection string along with URL property. Note that when configured with Kerberos, in order to participate in Kerberos based authentication you need to configure "user" property as required by "gss-pattern" or define the "authentication-type" property on the VDB or transport. However, after successful login into security-domain, the user name from GSS login context will be used for representing the session in the Teiid.

```
jaasName=Teiid;user={pattern};kerberosServicePrincipleName=host/testserver@MY_REALM
```

jassName defines the JAAS configuration name in login.config file. This property is optional, if omitted the "Teiid" is used as the default configuration name.

kerberosServicePrincipleName defines service principle that needs to be requested on behalf of the service that is being connected to using the Kerberos principle configured. If this property is omitted the default service principle would be "TEIID/hostname" and hostname is derived from the JDBC connection URL.

Note	In order to avoid adding the service principle name to all your JDBC and ODBC clients, Teiid can use the default service principle name as "TEIID/hostname". Create this service ticket in KDC. This also helps if you move your Teiid server one host to another by simply creating a new principle in KDC with new host name. Then you would only required to update hostname in the URL.

ODBC Client Configuration

Create a DSN for the VDB on the client machine to the VDB that you would like to connect using PostgreSQL ODBC driver. In order to participate in Kerberos based authentication you need to configure "user" property as required by "gss-pattern" or define the "authentication-type" property on the VDB or transport.

No additional configuration is needed as part of this, except that your workstation where the ODBC DSN exists must have been authenticated using GSS API against Active Directory or other Enterprise directory server. See this website http://spnego.sourceforge.net on instructions as to how to verify your system is authenticated into enterprise directory server. Contact your company’s operations team if you have any questions.

OData Client

The default OData client is configured with HTTP Basic authentication, to convert this authentication method into kerberos, clone or copy the maven project from https://github.com/teiid/teiid-web-security and then edit the web.xml and jboss-web.xml files and then replace MY_RELAM property with the property of security domain created above. Once the properties are updated, create a WAR file by running

```
mvn clean install
```

This will generate a new WAR file in "odata-kerberos/target" directory. Follow the below deployment direction based on your server.

Note	To use Kerberos or any web layer authentication, the OData war must use PassthroughAuthentication=true (which is the default).

Community Teiid Server based on WildFly

Replace the <wildfly>/modules/system/layers/dv/org/jboss/teiid/main/deployments/teiid-olingo-odata4.war" file with new WAR file, by executing a command similar to

{code} cp teiid-web-security/odata-kerberos/target/teiid-odata-kerberos-{version}.war <wildfly>/modules/system/layers/dv/org/jboss/teiid/main/deployments/teiid-olingo-odata4.war {code}

JDV Server

If you are working with JDV 6.3 server or greater, then run the following CLI script, you may have change the below script to adopt to the correct version of the WAR and directory names where the content is located.

```
undeploy teiid-olingo-odata4.war
deploy teiid-web-security/odata-kerberos/target/teiid-odata-kerberos-{version}.war
```

or overlay the new one using CLI script like

```
deployment-overlay add --name=myOverlay --content=/WEB-INF/web.xml=teiid-web-security/odata-kerberos/src/main/webapp/WEB-INF/web.xml,/WEB-INF/jboss-web.xml=teiid-web-security/odata-kerberos/src/main/webapp/WEB-INF/jboss-web.xml,/META-INF/MANIFEST.MF=teiid-web-security/odata-kerberos/src/main/webapp/META-INF/MANIFEST.MF --deployments=teiid-olingo-odata4.war --redeploy-affected
```

Custom Authorization Validator

In situations where Teiid's built-in Data Roles mechanism is not sufficient, a custom `org.teiid.PolicyDecider` can be installed via a JBoss module. Note that a PolicyDecider only makes high-level authorization decisions based upon the access context (INSERT, UPDATE, DELETE, etc.), the caller, and the resource (column, table/view, procedure, function, etc.). Data-level column masking and row based security policy information due to its interaction with the Teiid planner cannot be injected via a custom `org.teiid.PolicyDecider` . You may add column masking and row based security permissions via the `org.teiid.MetadataFactory` in custom a `org.teiid.MetadataRepository` or custom translator.

To provide a custom authorization validator, you must extend the `org.teiid.PolicyDecider` interface and build a custom java class. If you are using maven as your build process, you can use following dependencies:

```
<dependencies>
      <dependency>
         <groupId>org.teiid</groupId>
         <artifactId>teiid-api</artifactId>
         <scope>provided</scope>
      </dependency>
      <dependency>
         <groupId>org.teiid</groupId>
         <artifactId>teiid-common-core</artifactId>
         <scope>provided</scope>
      </dependency>
   </dependencies>
```

The *PoilcyDecider* interface is loaded by the Teiid using the Java's standard service loader mechanism. For this to work, add the following named file *META-INF/services/org.teiid.PolicyDecider* with full name of your PolicyDecider implementation class as its contents. for example:

META-INF/services/org.teiid.PolicyDecider

```
org.example.auth.MyCustomPolicyDecider
```

Now package all these files into a JAR archive file and build JBoss module in *jboss-as/modules* directory. If your PolicyDecider has any third party dependencies those jar files can also be added as dependencies to the same module. Make sure you list all the files in the *module.xml* file. Below is sample module.xml file along with Teiid specific dependencies

module.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<module xmlns="urn:jboss:module:1.0" name="org.example.auth">
    <resources>
        <resource-root path="my_custom_policy.jar" />
        <!--add any other dependent jars here, if they are not defined as modules -->
    </resources>


<dependencies>
    <module name="org.teiid.common-core"/>
    <module name="org.teiid.api"/>
    <module name="javax.api"/>
</dependencies>
</module>
```

create folder in the "<jboss-as>/modules/org/example/auth/main", copy the above *module.xml* file along with all the jar files. This directory can be different if you choose, just make sure the name of the module and the directory name match.

After the module has been added, change the configuration. Edit either the standalone-teiid.xml or te domain-teiid.xml file, and in the "teiid" subsystem xml fragment add the following xml with the module name created.

```
<policy-decider-module>name</policy-decider-module>
```

then restart the system. A `PolicyDecider` may be consulted many times for a single user command, but it is only called to make decisions based upon resources that appear in user queries. Any further access of resources through views or stored procedures, just as with data roles, is not checked against a `PolicyDecider` .

SAML Based Security For OData

By default the OData access to a Virtual Database (VDB) in WildFly is restricted to authentication using the HTTP Basic. However, it possible with below instructions one can configure OData access to participate in a Single-Sign-On (SSO) based security using SAML2. The below instructions are based on JBoss EAP platform using Picketlink security framework.

In SAML based authentication there are Identity Providers (IDP) who provide authentication services and Service Providers (SP), a end user service like odata and user (you). It is expected that you already have IDP, configured and working with security domain of your choice like LDAP or Kerberoes etc. The SP in this case is the OData WAR file that is supplied with Teiid distribution along with Picketlink based framework. Picketlink framework does not explicitly mention the interoperability with other third party external vendors supplied IDP, but Teiid team has tested successfully with

- Shibboleth
- Picketlink IDP
- Salesforce IDP (this is documented on Picketlink, not verified)
- Social Logins with Picketlink IDP (like, google, facebook etc. This has been mentioned in Picketlink documentation but not verified)

Note	Since SAML2 is standard, we believe any standards complaint IDP vendor will work with Picketlink SP.

requisites

- Collect the certificate for authentication that is used by IDP to sign the SAML messages.
- Gather the SSO POST based URL for your IDP, that your SP can use to redirect for authentication call.

Note	**"DNS Names"** - Do not try to use IP address or localhost except for the testing scenarios. Configure proper DNS names for both IDP and SP servers and make sure both can access each other using the URLs configured.

Configure for SAML based authentication the OData

In security-domains add following login module using the following CLI

```
/subsystem=security/security-domain=teiid-security/authentication=classic/login-module=RealmDirect:write-attribute(name=flag, value=sufficient)
/subsystem=security/security-domain=teiid-security/authentication=classic/login-module=saml2:add(code=org.picketlink.identity.federation.bindings.jboss.auth.SAML2LoginModule, flag=sufficient)
reload
```

the above commands will result in XML in standalone.xml or domain.xml file similar to:

"Security-Domain for SAML Authentication"

```
        <security-domain name="teiid-security">
        <authentication>
            <login-module code="org.picketlink.identity.federation.bindings.jboss.auth.SAML2LoginModule" flag="
sufficient"/>
            <login-module code="RealmDirect" flag="sufficient">
               <module-option name="password-stacking" value="useFirstPass"/>
```

```
            </login-module>
        </authentication>
    </security-domain>
```

Modify the OData WAR File to use SAML based authentication

- Extract the "teiid-olingo-odata4.war" file from "modules/system/base/dv/org/jboss/teiid/main/deployments" to another location. The WAR file is simple ZIP file so you can "jar -x teiid-olingo-odata4.war /modified"
- Edit "WEB-INF/jboss-web.xml" file, and it should look like

"jboss-web.xml"

```
<?xml version="1.0" encoding="UTF-8"?>
<jboss-web>
    <context-root>odata4</context-root>
    <security-domain>teiid-security</security-domain>
    <valve>
       <class-name>org.picketlink.identity.federation.bindings.tomcat.sp.ServiceProviderAuthenticator</class-na
me>
       <param>
      <param-name>configProvider</param-name>
      <param-value>org.picketlink.identity.federation.web.config.SPPostMetadataConfigurationProvider</param-val
ue>
    </param>
    </valve>
</jboss-web>
```

- Edit "web.xml" file and **remove** the section below

"web.xml"

```
<login-config>
    <auth-method>BASIC</auth-method>
    <realm-name>yourdomain.com</realm-name>
</login-config>
```

- Add the certificate keystore from your IDP to the classes directory. This is {KEYSTORE-FILE} in below configuration. or you can add to a existing keystore using following command

```
keytool -import -file idp_cert.cer -keystore {KEYSTORE-FILE} -alias {CERTIFICATE-
ALIAS}
```

- Add "picketlink.xml" file to WEB-INF directory with following content

"picketlink.xml"

```
<PicketLink xmlns="urn:picketlink:identity-federation:config:2.1">
    <PicketLinkSP xmlns="urn:picketlink:identity-federation:config:2.1"
        ServerEnvironment="tomcat" BindingType="POST" SupportsSignatures="true">
        <KeyProvider
            ClassName="org.picketlink.identity.federation.core.impl.KeyStoreKeyManager">
            <Auth Key="KeyStoreURL" Value="{KEYSTORE-FILE}" />
            <Auth Key="KeyStorePass" Value="{KEYSTORE-PASSWORD}" />
            <Auth Key="SigningKeyAlias" Value="{CERTIFICATE-ALIAS}" />
                        <Auth Key="SigningKeyPass" Value="{CERTIFICATE-PASSWORD}" />
            <ValidatingAlias Key="localhost" Value="{CERTIFICATE-ALIAS}" />
            <ValidatingAlias Key="127.0.0.1" Value="{CERTIFICATE-ALIAS}" />
        </KeyProvider>
    </PicketLinkSP>
    <Handlers xmlns="urn:picketlink:identity-federation:handler:config:2.1">
```

```
        <Handler class="org.picketlink.identity.federation.web.handlers.saml2.SAML2LogOutHandler" />
        <Handler class="org.picketlink.identity.federation.web.handlers.saml2.SAML2AuthenticationHandler" />
        <Handler class="org.picketlink.identity.federation.web.handlers.saml2.RolesGenerationHandler" />
        <Handler class="org.picketlink.identity.federation.web.handlers.saml2.SAML2SignatureGenerationHandler"
/>
        <Handler class="org.picketlink.identity.federation.web.handlers.saml2.SAML2SignatureValidationHandler"
/>
    </Handlers>
</PicketLink>
```

Note	{CERTIFICATE-ALIAS} is typically something like "idp.example.com" for which the certificate is created for

- Add the certificate received from IDP vendor to "WEB-INF/classes" directory. Note this must be same name as {CERTIFICATE-FILE-NAME} used in "Configuring the Picketlink Subsystem"
- Add "sp-metadata.xml" to the classes directory. Note that your "sp-metadata.xml" contents will entirely dependent upon your Identity Provider settings. The below sample **ONLY** provided as an example

"sp-metadata.xml"

```
<?xml version="1.0" encoding="UTF-8"?>
<EntitiesDescriptor Name="urn:mace:shibboleth:testshib:two"
    xmlns:shibmd="urn:mace:shibboleth:metadata:1.0" xmlns="urn:oasis:names:tc:SAML:2.0:metadata"
    xmlns:ds="http://www.w3.org/2000/09/xmldsig#" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <EntityDescriptor entityID="http://localhost:8080/idp-metadata/">
        <IDPSSODescriptor
            protocolSupportEnumeration="urn:oasis:names:tc:SAML:1.1:protocol urn:oasis:names:tc:SAML:2.0:protoc
ol">
            <NameIDFormat>urn:oasis:names:tc:SAML:2.0:nameid-format:transient
            </NameIDFormat>
            <SingleSignOnService Binding="urn:mace:shibboleth:1.0:profiles:AuthnRequest"
                Location="http://localhost:8080/idp-metadata/" />
            <SingleSignOnService Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-POST"
                Location="http://localhost:8080/idp-metadata/" />
            <SingleSignOnService
                Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-Redirect"
                Location="http://localhost:8080/idp-metadata/" />
            <SingleLogoutService
                Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-POST"
                Location="http://localhost:8080/idp-metadata/?GLO=true" />
            <SingleLogoutService
                Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-Redirect"
                Location="http://localhost:8080/idp-metadata/SLO" />
        </IDPSSODescriptor>
        <Organization>
            <OrganizationName xmlns="urn:oasis:names:tc:SAML:2.0:metadata"
                xml:lang="en">JBoss</OrganizationName>
            <OrganizationDisplayName xmlns="urn:oasis:names:tc:SAML:2.0:metadata"
                xml:lang="en">JBoss by Red Hat</OrganizationDisplayName>
            <OrganizationURL xmlns="urn:oasis:names:tc:SAML:2.0:metadata"
                xml:lang="en">http://www.jboss.org</OrganizationURL>
        </Organization>
        <ContactPerson contactType="technical">
            <GivenName>The</GivenName>
            <SurName>Admin</SurName>
            <EmailAddress>admin@mycompany.com</EmailAddress>
        </ContactPerson>
    </EntityDescriptor>
    <EntityDescriptor entityID="http://localhost:8080/odata4/">
        <SPSSODescriptor
            protocolSupportEnumeration="urn:oasis:names:tc:SAML:2.0:protocol urn:oasis:names:tc:SAML:1.1:protoc
ol http://schemas.xmlsoap.org/ws/2003/07/secext">
            <NameIDFormat>urn:oasis:names:tc:SAML:2.0:nameid-format:transient
            </NameIDFormat>
            <AssertionConsumerService
                Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-POST" Location="http://localhost:8080/odata4
/"
```

```
                index="1" isDefault="true" />
        </SPSSODescriptor>
        <Organization>
            <OrganizationName xmlns="urn:oasis:names:tc:SAML:2.0:metadata"
                xml:lang="en">JBoss</OrganizationName>
            <OrganizationDisplayName xmlns="urn:oasis:names:tc:SAML:2.0:metadata"
                xml:lang="en">JBoss by Red Hat</OrganizationDisplayName>
            <OrganizationURL xmlns="urn:oasis:names:tc:SAML:2.0:metadata"
                xml:lang="en">http://localhost:8080/odata4/</OrganizationURL>
        </Organization>
        <ContactPerson contactType="technical">
            <GivenName>The</GivenName>
            <SurName>Admin</SurName>
            <EmailAddress>admin@mycompany.com</EmailAddress>
        </ContactPerson>
    </EntityDescriptor>
</EntitiesDescriptor>
```

- Create a deployment-overlay using the cli with the modified contents:

```
deployment-overlay add --name=myOverlay --content=/WEB-INF/web.xml=/modified/web.xml,/WEB-INF/jboss-web.xml=/mo
dified/jboss-web.xml --deployments=teiid-odata-odata4.war --redeploy-affected
```

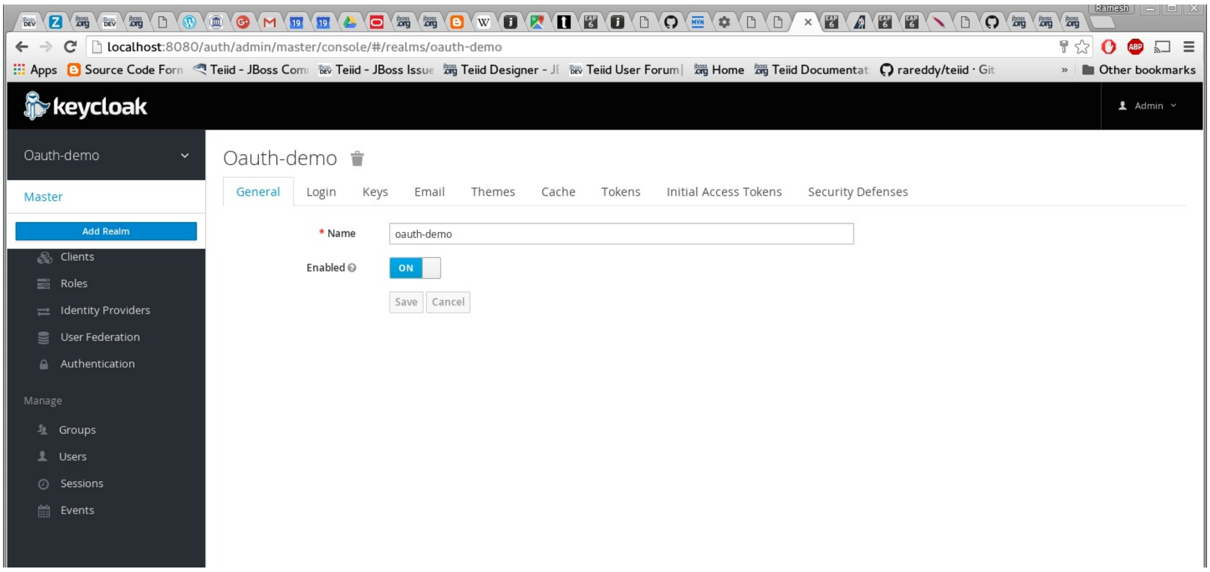
OAuth2 Based Security For OData Using KeyCloak

This document will provide detailed instructions to enable OAuth V2 authentication on Teiid’s OData interface using the Keycloak as authentication server (IDP). Please note that use a different IDP server will not work with this implementation as OAuth2 implementations are not interoperable. To work with separate IDP than Keycloak consult their documentation, replace the web layer semantics, like the "login-config" in web.xml file etc. Providing the details of other IDP is beyond the scope of this document.

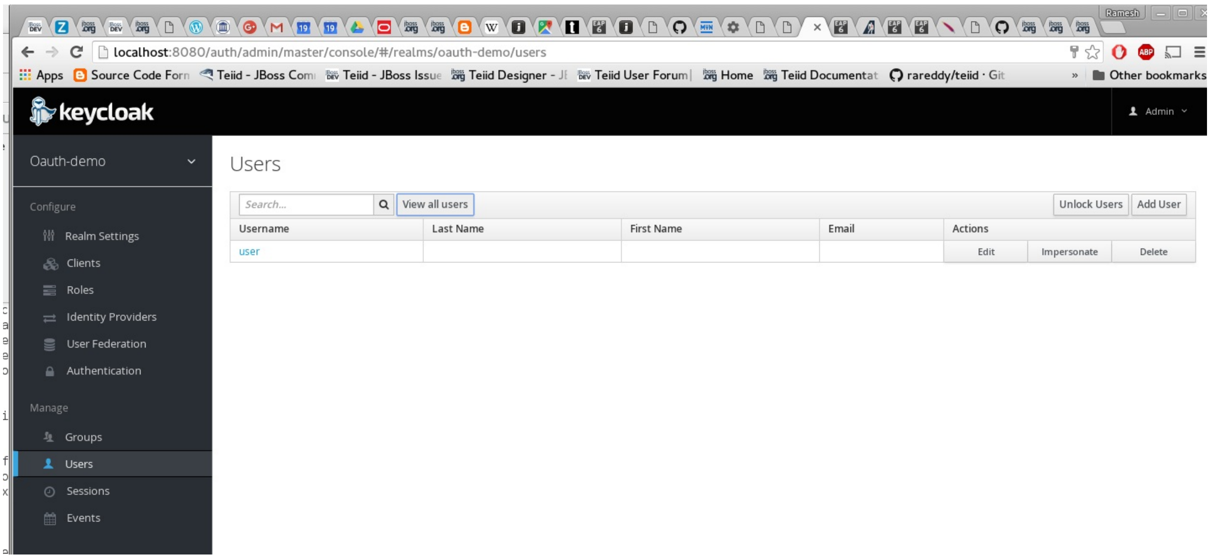
This examples will show case an example, where Teiid’s OData rest interface is secured using OAuth using Keycloak using OpenID Connect. The VDB accessed by the OData interface also depends on another web service which is used as a data source, that is also secured with OAuth using the same Keycloak IDP. The central idea behind this example is to pass the same "access-token" used at OData interface layer to passthrough the Teiid layer to bottom data source layer and gain access to the source.

Download and install Keycloak as a separate web server.

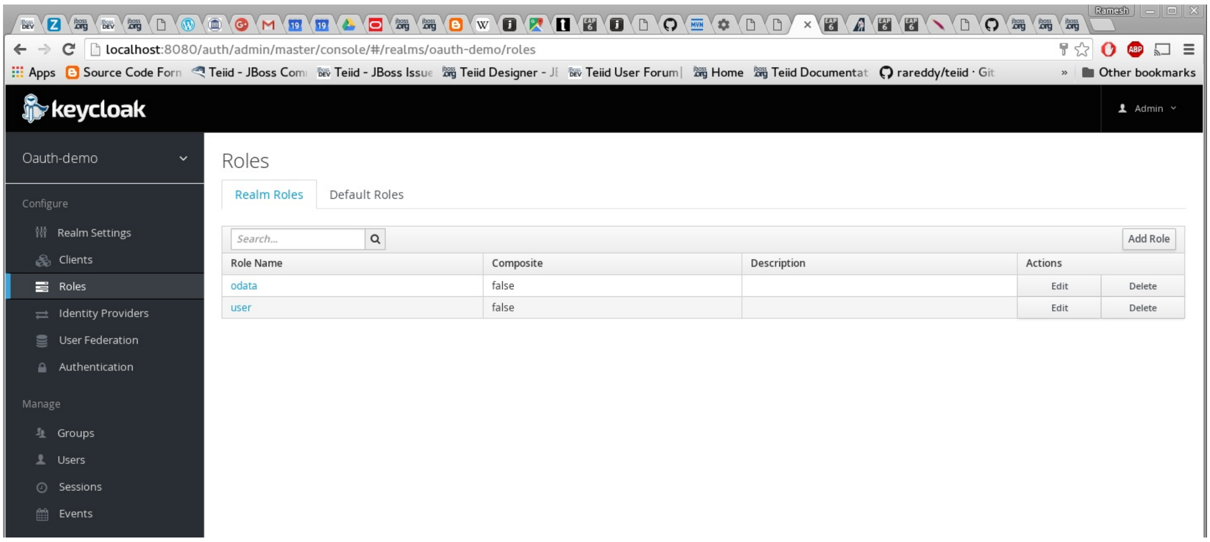
- Login using the default "admin/admin" credentials into the Keycloak "master" realm.
- Add a new realm called "oauth-demo"



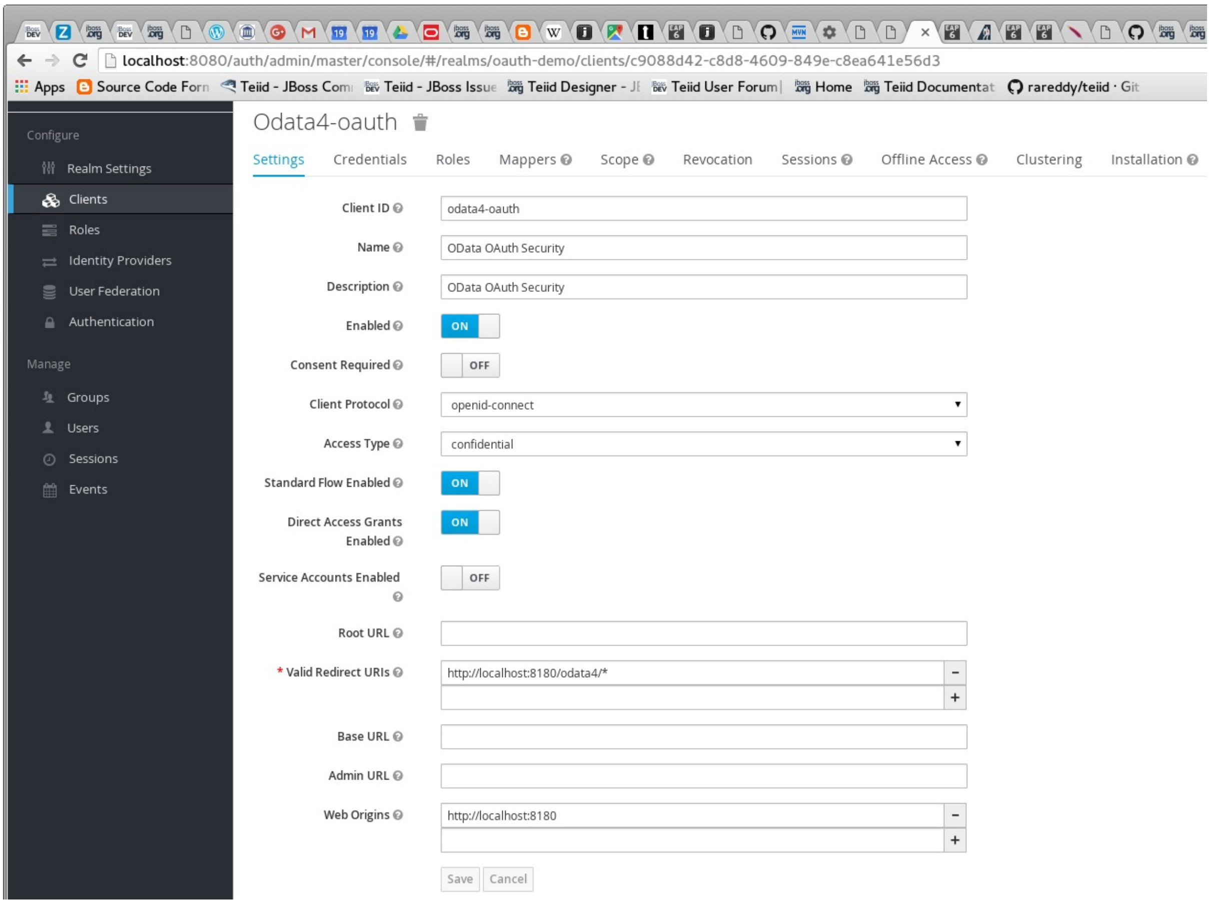
- Add a new user called "user" and add credentials.



Add two roles "odata" and "user". These are enterprise roles, that will be used by the web services to grant the access to user. Also these roles are used as "scopes" in the OAuth protocol.

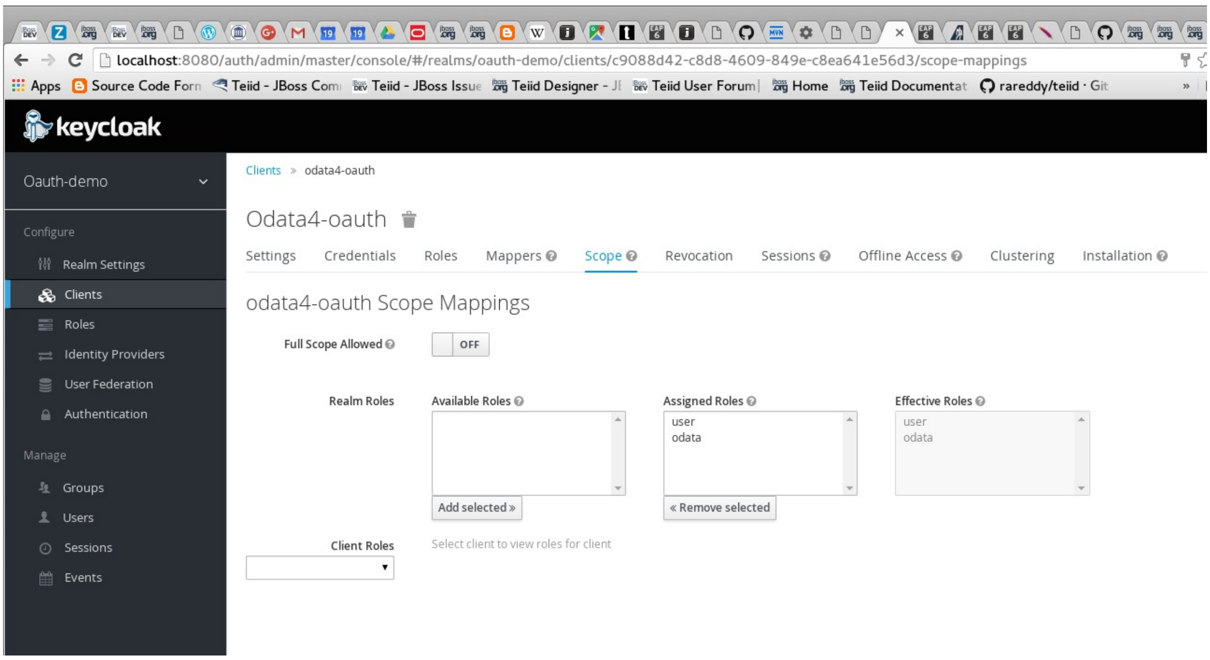


- Add a new client called "odata4-oauth", this client represents the Teiid’s OData client that we are going to create

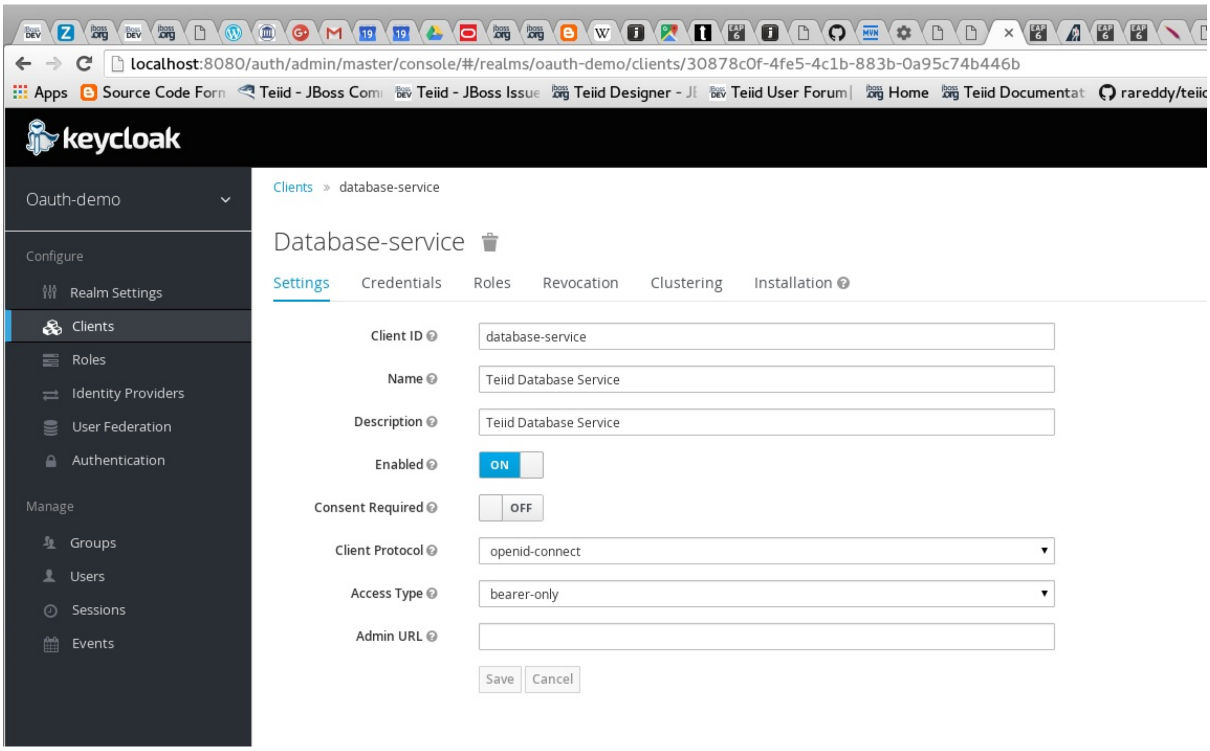


and choose scopes "odata" and "user" for this client. Note that the redirect URI needs to be where the actual service is going to be available.

Note	The client web-service typically defines what roles that logged in user must have in order for to grant the access. In the Keycloak OAuth implementation, these roles are used as "scopes". Note that the "odata4-oauth" client MUST have ALL the scopes that it is going to delegate the access-token for gaining access to bottom data services. In this example Teiid's OData web services requires "odata" role, the bottom web-service requires the "user" role. Since the OData accesses the bottom web-service it requires both the roles.



- Add another client called "database-service" and choose scope "user". Choose type as "Bearer".



Install and configure Teiid server

- Download and install Teiid server
- Download Keycloak adapter for the EAP, and unzip over the Teiid server installation or follow Keycloak installtion directions.

Run the following to add Keycloak specific modules to the server

```
$ cd $WILDFLY_HOME
$ unzip keycloak-wildfly-adapter-dist-${version}.zip
```

Now, start the Teiid Server and using the jboss-cli.sh file run the following to install the KeyCloak configuration into the Teiid Server.

```
./bin/jboss-cli.sh --file=adapter-install.cli
```

Then we need to change the OData transport’s "security-domain" to "keycloak".

```
./bin/jboss-cli.sh --connect
/subsystem=teiid:write-attribute(name=authentication-security-domain, value=keycloak)

reload
```

above commands will result in XML in standalone.xml or domain.xml file like (you can also edit standalone.xml directly)

```
      <authentication security-domain="keycloak"/>
      <transport name="odata"/>
```

The Keycloak is installed and the OData transport is modified, now we need to install security-domain called "passthrough". Note that the web layer is using OAuth2, but at the VDB layer, this logged in user need to be passed through and this security domain will help with that.

```
./bin/jboss-cli.sh --connect
/subsystem=security/security-domain=passthrough:add(cache-type=default)
/subsystem=security/security-domain=passthrough/authentication=classic:add
/subsystem=security/security-domain=passthrough/authentication=classic/login-module=passthrough:add(code=org.te
iid.jboss.PassthroughIdentityLoginModule, flag=required, module=org.jboss.teiid)

reload
```

above commands will result in XML in standalone.xml or domain.xml file like (you can also edit standalone.xml directly)

```
    <security-domain name="passthrough">
        <authentication>
            <login-module code="org.teiid.jboss.PassthroughIdentityLoginModule" flag="required" module="org.jbo
ss.teiid"/>
        </authentication>
    </security-domain>
```

This finishes all the server side changes that are required to make OAuth authentication using Keycloak.

OData Application WAR

In order to use OAuth2 authentication, the OData WAR needs to be updated to make use of the OAuth2 based security domain. By default Teiid installation comes with OData web service WAR file configured with "HTTP Basic" authentication. This WAR needs to either replaced or updated.

Build the new OData WAR file that supports OAuth.

To build OAuth based OData WAR file, Teiid provides a template maven project, either download or clone the project from https://github.com/teiid/teiid-web-security

The above link provides templates for creating two WAR files, one WAR file is to create Teiid’s OData service with OAuth, the next is a sample "database-service" for this demo. Please note that "database-service" is to mimic the database service, that will be different in a real use-case, however the steps defined for the access will be same.

Replace the "teiid-web-security/teiid-odata-oauth-keycloak/src/main/webapp/WEB-INF/keyclock.json" file contents with "installation" script in "keycloak.json" format from Keycloak admin console’s "odata4-client" client application.

Similarly replace the "teiid-web-security/examples/database-service/src/main/webapp/WEB-INF/keyclock.json" file contents with "installation" script in "keycloak.json" format from Keycloak admin console’s "database-client" client application.

Edit the "teiid-web-security/odata-oauth-keycloak/src/main/webapp/WEB-INF/web.xml" file to enable Passthrough Authentication

```
    <init-param>
        <param-name>PassthroughAuthentication</param-name>
        <param-value>true</param-value>
    </init-param>
```

Build the WAR files running the maven command

```
mvn clean package
```

Note	You may have to update Teiid and Keycloak versions in the pom.xml file

The above command will generate a new WAR file for deployment. Follow the below directions to deploy this new WAR file.

Teiid Server on WildFly

Replace the <wildfly>/modules/system/layers/dv/org/jboss/teiid/main/deployments/teiid-olingo-odata4.war" file with new WAR file, by executing a command similar to

```
cp teiid-web-security/odata-oauth-keycloak/target/teiid-odata-oauth-keycloak-
{version}.war \
   <wildfly>/modules/system/layers/dv/org/jboss/teiid/main/deployments/teiid-
olingo-odata4.war
```

JDV Server

If you are working with JDV 6.3 server or greater, then run the following CLI script, you may have change the below script to adopt to the correct version of the WAR and directory names where the content is located.

```
undeploy teiid-olingo-odata4.war
deploy teiid-web-security/odata-oauth-keycloak/target/teiid-odata-oauth-keycloak-
{version}.war
```

or overlay the new one using CLI script like

```
deployment-overlay add --name=myOverlay --content=/WEB-INF/web.xml=teiid-web-
security/odata-oauth-keycloak/src/main/webapp/WEB-INF/web.xml,/WEB-INF/jboss-
web.xml=teiid-web-security/odata-oauth-keycloak/src/main/webapp/WEB-INF/jboss-
web.xml,/META-INF/MANIFEST.MF=teiid-web-security/odata-oauth-
keycloak/src/main/webapp/META-INF/MANIFEST.MF,/WEB-INF/keycloak.json=teiid-web-
security/odata-oauth-keycloak/src/main/webapp/WEB-INF/keycloak.json /WEB-
INF/lib/teiid-odata-oauth-keycloak-{version}.jar=teiid-web-security/odata-oauth-
keycloak/src/main/webapp/WEB-INF/lib/teiid-odata-oauth-keycloak-{version}.jar --
deployments=teiid-olingo-odata4.war --redeploy-affected
```

Working with example VDB

```
<vdb name="oauthdemo" version="1">
    <model visible="true" name="PM1">
        <source name="any" translator-name="loopback"/>
        <metadata type = "DDL"><![CDATA[
            CREATE FOREIGN TABLE G1 (e1 integer PRIMARY KEY, e2 varchar(25), e3 double);
        ]]>
        </metadata>
    </model>
</vdb>
```

Start both Keycloak and Teiid Servers. If both of these servers are in the same machine, then we need to offset the ports of Teiid server such that they will not conflict with that of the Keycloak server. For this example, I started the Teiid server as

```
./standalone.sh -c standalone-teiid.xml -Djboss.socket.binding.port-offset=100
```

where all ports are offset by 100. So the management port is 10090 and default JDBC port will be 31100. The Keycloak server is started on default ports.

Testing the example

There are two different mechanisms for testing this example. One is purely for testing the using the browser, then other is programatically. Typically using the browser is NOT correct for accessing the Teiid's OData service, but it is shown below for testing purposes.

Using the Web Browser

Using the browser issue a query (the use of browser is needed because, this process does few redirects only browsers can automatically follow)

```
http://localhost:8180/odata4/oauthdemo/PM1/G1
```

The user will be presented with Keycloak based login page, once the credentials are presented the results of the above request are shown.

Calling programatically

This process of calling does not need to involve a web-browser, this is typical of scenario where another web-application or mobile application is calling the Teiid's OData web-service to retrieve the data. However in this process, the process of negotiating the "access-token" is externalized and is defined by the IDP, which in this case is Keycloak.

For demonstration purposes we can use CURL to negotiate this token as shown below (client_secret can found the Keycloak admin console under client credentials tab)

```
curl -v POST http://localhost:8080/auth/realms/oauth-demo/protocol/openid-connect/token  -H "Content-Type: appl
ication/x-www-form-urlencoded" -d 'username=user' -d 'password=user' -d 'grant_type=password' -d 'client_id=oda
ta4-oauth' -d 'client_secret=36fdc2b9-d2d3-48df-8eea-99c0e729f525'
```

this should return a JSON payload similar to

```
{  "access_token":"eyJhbGciOiJSUzI1NiJ9.eyJqdGkiOiI0YjI4NDMzYS1..",
   "expires_in":300,
   "refresh_expires_in":1800,
   "refresh_token":"eyJhbGciOiJSUzI1NiJ9.eyJqdGkiOiJmY2JmNjY2ZC0xNzIwLTQwODQtOTBiMi0wMjg4ODdhNDkyZWYiLCJl..",
   "token_type":"bearer",
   "id_token":"eyJhbGciOiJSUzI1NiJ9.eyJqdGkiOiIwZjYyNDQ1MS1iNTE0LTQ5YjUtODZlNy1jNTI5MDU2OTI3ZDIiLCJleH..",
   "not-before-policy":0,
   "session-state":"6c8884e8-c5aa-4f7a-a3fe-9a7f6c32658c"
}
```

from the above you can take the "access_token" and issue the query to fetch results like

```
curl -k -H "Authorization: Bearer eyJhbGciOiJSUzI1NiJ9.eyJqdGkiOiI0YjI4NDMzYS1.." http://localhost:8180/odata4/
oauthdemo/PM1/G1
```

You should see same XML response as above. Please note that to programatically achieve the access_token in your own program (not using curl) you can see some suggestions in this document [https://keycloak.gitbooks.io/documentation/server_development/topics/admin-rest-api.html]

SAML Based Security For OData Using KeyCloak

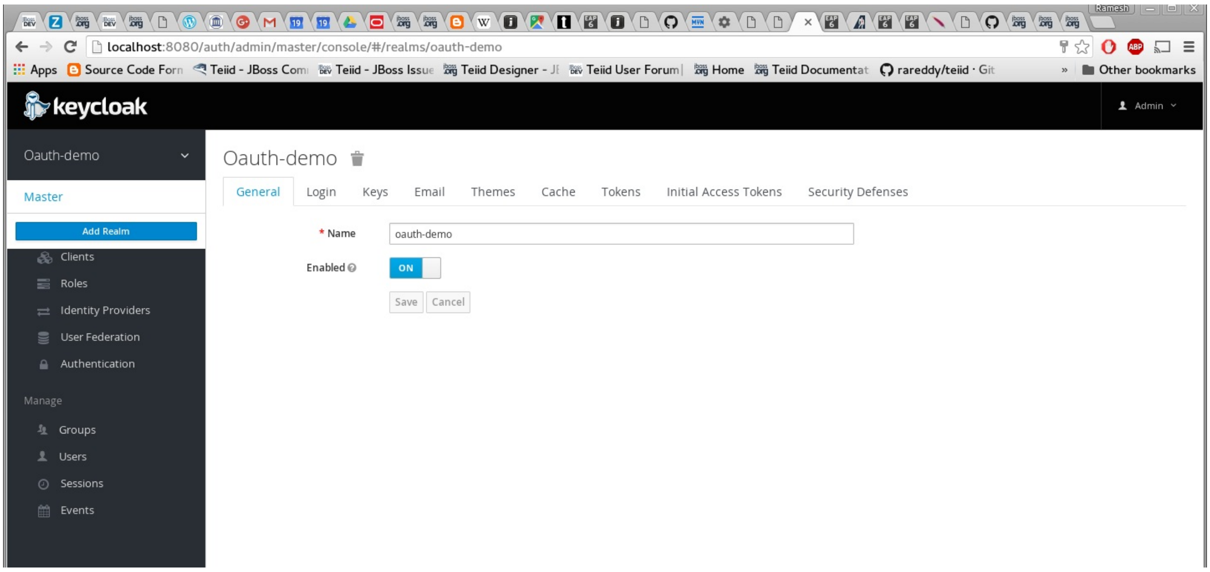
This document will provide detailed instructions to enable SAML authentication on Teiid's OData interface using the Keycloak as authentication server (IDP). SAML is standard, so the modified OData WAR should work fine with any other compatible SAML Authorization server, however the configuration may be little different. Please consult their documentation for any such specifics of different authorization server other then KeyCloak.

This examples will show case an example, where Teiid's OData rest interface is secured using SAML using Keycloak as IDP. The VDB accessed by the OData interface, the pass-through of SAML Assertion for OAuth token (SAML Bearer) is not yet available in KeyCloak, when the feature is available then Teiid will support it. However, if you are working with a IDP that supports the SAML Bearer, Teiid does support the mechanism where one can pass the "access-token" from web layer to the data source layer. See the OAuth example as template and possible configuration needed. (note it is not exactly same, but very similar)

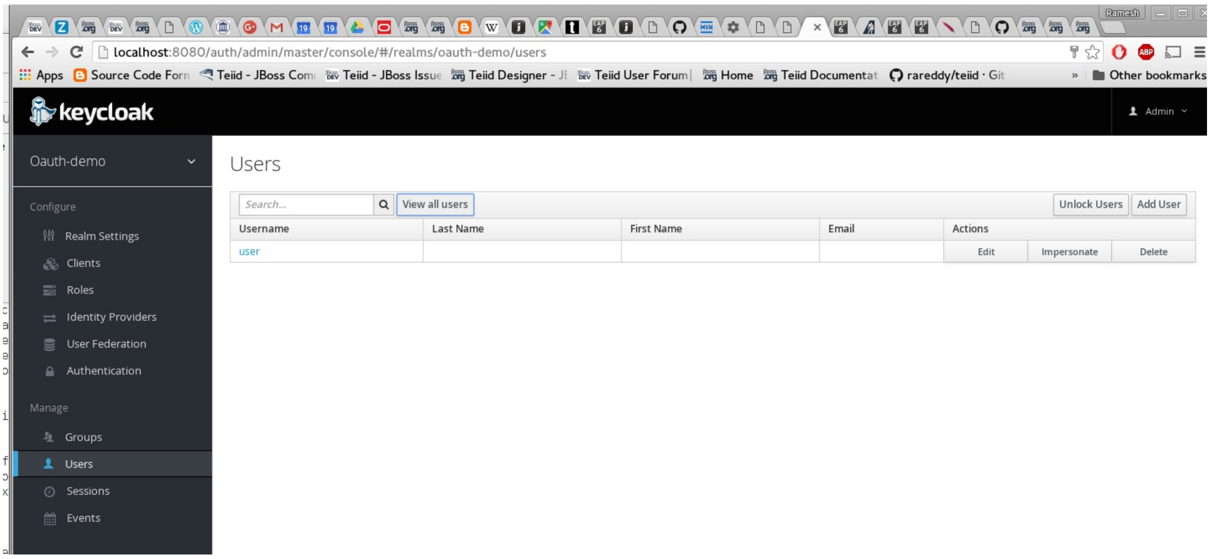
Tested with Keycloak 3.1.0.Final version.

Download and install Keycloak as a separate web server.

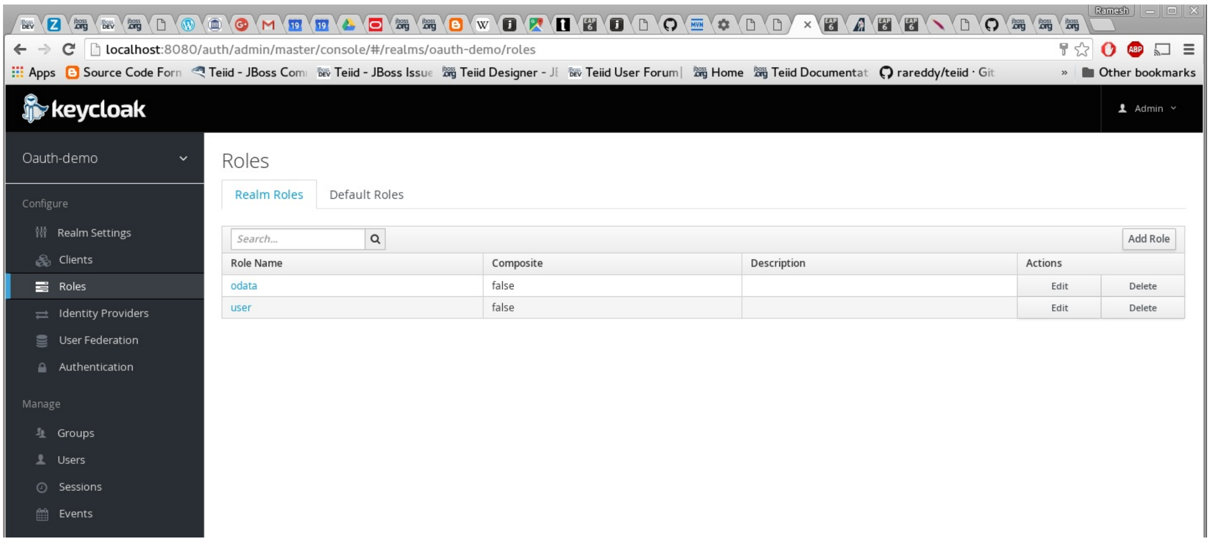
- Login using the default "admin/admin" credentials into the Keycloak "master" realm.
- Add a new realm called "oauth-demo"



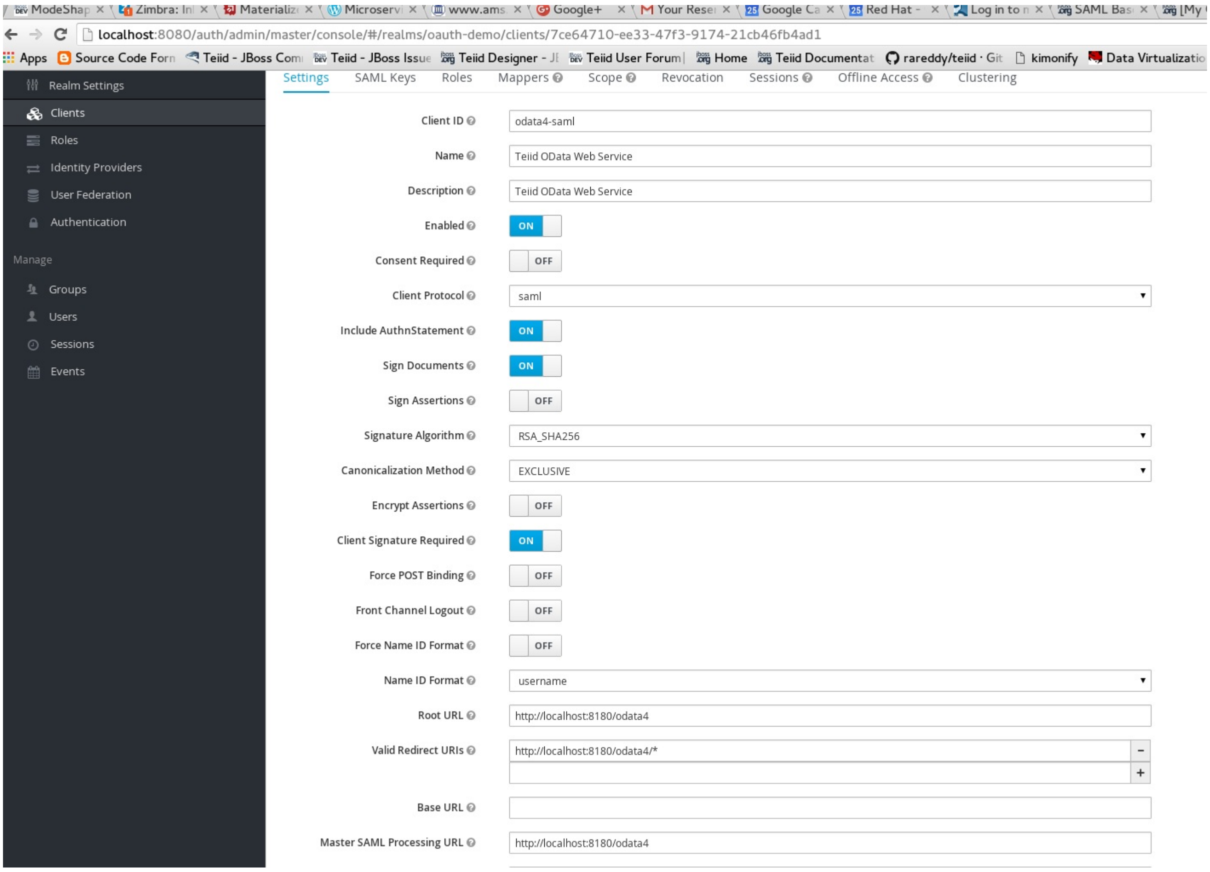
- Add a new user called "user" and add credentials.



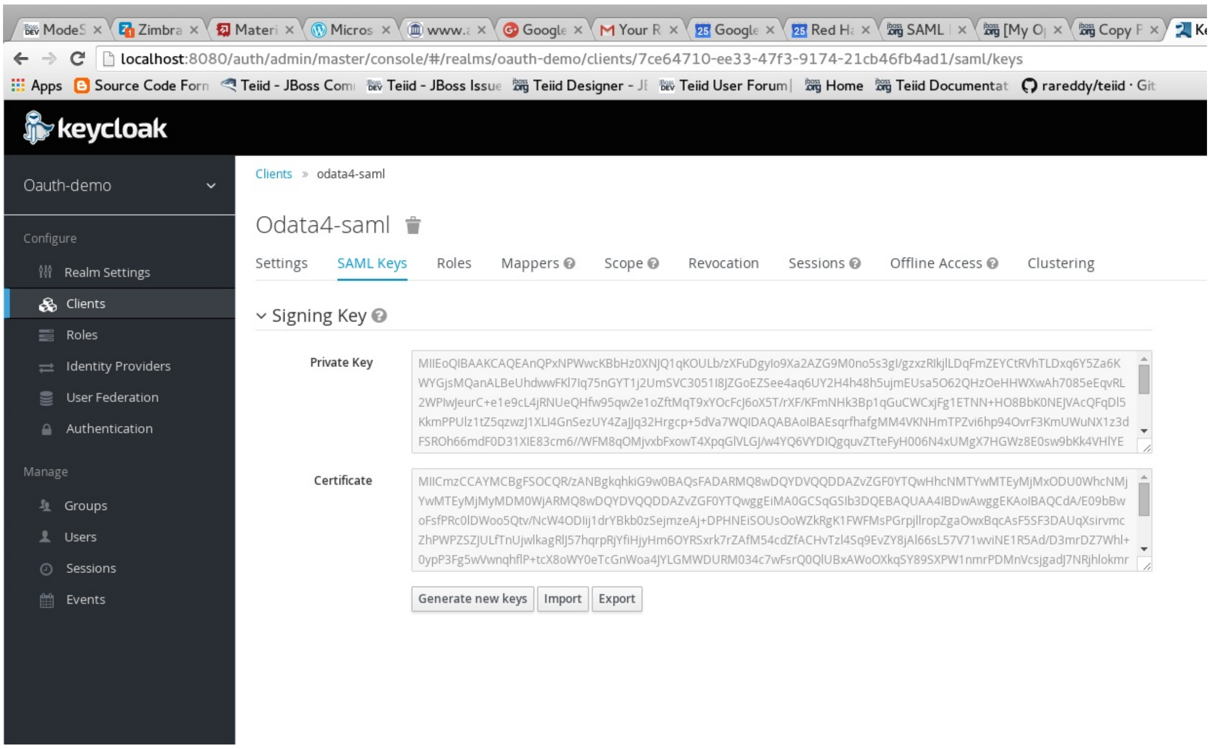
Add two roles "odata" and "user". These are enterprise roles, that will be used by the web services to grant the access to user. Also these roles are used as "scopes" in the OAuth protocol.



- Add a new client called "odata4-saml", this client represents the Teiid’s SAML client that we are going to create



- Click on SAML Keys, either import your certificate or generate a new one. Then click export, and keep the exported certificate for later use.



Install and configure Teiid server

- Download and install Teiid server
- Download Keycloak SAML adapter for EAP, and unzip over the Teiid server installation.

Run the following to add Keycloak specific modules to the server

```
$ cd $WILDFLY_HOME
$ unzip keycloak-saml-wildfly-adapter-dist-${version}.zip
```

Now, start the Teiid Server and using the jboss-cli.sh file run the following to install the KeyCloak confiuration into the Teiid Server.

```
./bin/jboss-cli.sh --file=adapter-install-saml.cli
```

In security-domains add following login module using the following CLI

```
subsystem=security/security-domain=saml-security/authentication=classic/login-
module=RealmDirect:write-attribute(name=flag, value=sufficient)
/subsystem=security/security-domain=saml-security/authentication=classic/login-
module=keycloak:add(code=org.keycloak.adapters.jboss.KeycloakLoginModule,
flag=sufficient)
reload
```

the above commands will result in XML in standalone.xml or domain.xml file like similar to:

```
    <security-domain name="saml-security">
        <authentication>
            <login-module code="org.keycloak.adapters.jboss.KeycloakLoginModule" flag="sufficient"/>
            <login-module code="RealmDirect" flag="sufficient">
               <module-option name="password-stacking" value="useFirstPass"/>
            </login-module>
        </authentication>
    </security-domain>
```

This finishes all the server side changes that are required to make SAML authentication using Keycloak.

OData Application WAR

In order to use SAML authentication, the OData WAR needs to be updated to make use of the OAuth based security domain. By default Teiid installation comes with OData web service WAR file configured with "HTTP Basic" authentication. This WAR needs to either replaced or updated.

Build the new OData WAR file that supports SAML.

- To build SAML based OData WAR file, Teiid provides a template maven project, either download or clone the project from https://github.com/teiid/teiid-web-security
- The above link provides templates for creating two WAR files, one WAR file is to create Teiid's OData service with OAuth, the next is for SAML. Choose the SAML one.
- Replace the "teiid-web-security/teiid-odata-saml-keycloak/src/main/webapp/WEB-INF/keyclock.json" file contents with "installation" script in "keycloak.json" format from Keycloak admin console's "odata4-saml" client application.

- Similarly replace the "teiid-web-security/teiid-odata-saml-keycloak/src/main/webapp/WEB-INF/keystore.jks" file with the exported keystore from earlier steps.
- build the "keycloak-saml.xml" file, and add all the sections of "metadata" specific to your service. This is where service knows where IDP located and which service this represents etc.

The build the WAR files running the maven command

```
mvn clean package
```

Note	You may have to update Teiid and Keycloak versions in the pom.xml file

The above command will generate a new WAR file for deployment. Follow the below directions to deploy this new WAR file to the server

Community Teiid Server on Wildfly

Replace the <wildfly>/modules/system/layers/dv/org/jboss/teiid/main/deployments/teiid-olingo-odata4.war" file with new WAR file, by executing a command similar to

```
cp teiid-web-security/teiid-odata-saml-keycloak/target/teiid-odata-saml-keycloak-
{version}.war
<wildfly>/modules/system/layers/dv/org/jboss/teiid/main/deployments/teiid-olingo-
odata4.war
```

JDV Server

If you are working with JDV 6.3 server or greater, then run the following CLI script, you may have change the below script to adopt to the correct version of the WAR and directory names where the content is located.

```
undeploy teiid-olingo-odata4.war
deploy teiid-web-security/teiid-odata-saml-keycloak/target/teiid-odata-saml-
keycloak-{version}.war
```

or overlay the new one using CLI script like

```
deployment-overlay add --name=myOverlay --content=/WEB-INF/web.xml=teiid-web-
security/teiid-odata-saml-keycloak/src/main/webapp/WEB-INF/web.xml,/WEB-INF/jboss-
web.xml=teiid-web-security/teiid-odata-saml-keycloak/src/main/webapp/WEB-INF/jboss-
web.xml,/META-INF/MANIFEST.MF=teiid-web-security/teiid-odata-saml-
keycloak/src/main/webapp/META-INF/MANIFEST.MF,/WEB-INF/keycloak-saml.xml=teiid-web-
security/teiid-odata-saml-keycloak/src/main/webapp/WEB-INF/keycloak-saml.xml,/WEB-
INF/keycloak.jks=teiid-web-security/teiid-odata-saml-keycloak/src/main/webapp/WEB-
INF/keycloak.jks --deployments=teiid-olingo-odata4.war --redeploy-affected
```

In the VDB, define the security layer for the VDB as "saml-security", for example

```
<vdb name="samldemo" version="1">
    <property name="security-domain" value="saml-security"/>
    <model visible="true" name="PM1">
```

```
        <source name="any" translator-name="loopback"/>
        <metadata type = "DDL"><![CDATA[
            CREATE FOREIGN TABLE G1 (e1 integer PRIMARY KEY, e2 varchar(25), e3
double);
        ]]>
       </metadata>
    </model>
</vdb>
```

Testing the example using Web Browser

To test any SAML based application you must use a Web browser. Using a browser issue any OData specific query, and you will be redirected to do SAML authentication.

```
http://localhost:8180/odata4/<vdb>.<version>/<model>/<view>
```