Table of Contents

- Introduction 1.1
- Legal Notice 1.2
- Administrator’s Guide 1.3
  - Installation Guide 1.3.1
  - Deploying VDBs 1.3.2
    - Deploying VDB Dependencies 1.3.2.1
      - Accumulo Data Sources 1.3.2.1.1
      - Amazon SimpleDB Data Sources 1.3.2.1.2
      - Cassandra Data Sources 1.3.2.1.3
      - File Data Sources 1.3.2.1.4
      - Ftp/Ftps Data Sources 1.3.2.1.5
      - Google Spreadsheet Data Sources 1.3.2.1.6
      - JDG Library Mode Data Sources 1.3.2.1.7
      - JDG HotRod Data Sources 1.3.2.1.8
      - JDBC Data Sources 1.3.2.1.9
      - LDAP Data Sources 1.3.2.1.10
      - MongoDB Data Sources 1.3.2.1.11
      - Phoenix Data Sources 1.3.2.1.12
      - Salesforce Data Sources 1.3.2.1.13
      - Solr Data Sources 1.3.2.1.14
      - Web Service Data Sources 1.3.2.1.15
        - Kerberos with REST based Services 1.3.2.1.15.1
        - OAuth Authentication With REST Based Services 1.3.2.1.15.2
    - VDB Versioning 1.3.2.2
  - Logging 1.3.3
  - Clustering in Teiid 1.3.4
  - Monitoring 1.3.5
  - Performance Tuning 1.3.6
    - Memory Management 1.3.6.1
    - Threading 1.3.6.2
    - Cache Tuning 1.3.6.3
    - Socket Transports 1.3.6.4
    - LOBs 1.3.6.5
    - Other Considerations 1.3.6.6
  - Teiid Console 1.3.7
  - AdminShell 1.3.8
    - Getting Started 1.3.8.1

Executing a script file 1.3.8.2
Log File and Recorded Script file 1.3.8.3
Default Connection Properties 1.3.8.4
Handling Multiple Connections 1.3.8.5
Interactive Shell Nuances 1.3.8.6
Other Scripting Environments 1.3.9
System Properties 1.3.10
Teiid Management CLI 1.3.11
Diagnosing Issues 1.3.12
Migration Guide From Teiid 8.x 1.3.13
Caching Guide 1.4
Results Caching 1.4.1
Materialized Views 1.4.2
External Materialization 1.4.2.1
Internal Materialization 1.4.2.2
Code Table Caching 1.4.3
Translator Results Caching 1.4.4
Hints and Options 1.4.5
Programmatic Control 1.4.6
Client Developer’s Guide 1.5
JDBC Support 1.5.1
Connecting to a Teiid Server 1.5.1.1
Driver Connection 1.5.1.1.1
DataSource Connection 1.5.1.1.2
Standalone Application 1.5.1.1.3
WildFly DataSource 1.5.1.1.4
Using Multiple Hosts 1.5.1.1.5
SSL Client Connections 1.5.1.1.6
Additional Socket Client Settings 1.5.1.1.7
Prepared Statements 1.5.1.2
ResultSet Limitations 1.5.1.3
JDBC Extensions 1.5.1.4
Statement Extensions 1.5.1.4.1
Partial Results Mode 1.5.1.4.2
Non-blocking Statement Execution 1.5.1.4.3
ResultSet Extensions 1.5.1.4.4
Connection Extensions 1.5.1.4.5
Unsupported JDBC Methods 1.5.1.5
Unsupported Classes and Methods in "java.sql" 1.5.1.5.1
Unsupported Classes and Methods in "javax.sql" 1.5.1.5.2

ODBC Support 1.5.2
Installing the ODBC Driver Client 1.5.2.1
Configuring the Data Source Name (DSN) 1.5.2.2
DSN Less Connection 1.5.2.3
ODBC Connection Properties 1.5.2.4
OData Support 1.5.3
OData Version 4.0 Support 1.5.3.1
Using Teiid with Hibernate 1.5.4
Using Teiid with EclipseLink 1.5.5
GeoServer Integration 1.5.6
QGIS Integration 1.5.7
Reauthentication 1.5.8
Execution Properties 1.5.9
XML extensions 1.5.9.1
SET Statement 1.5.10
SHOW Statement 1.5.11
Transactions 1.5.12
Local Transactions 1.5.12.1
Request Level Transactions 1.5.12.2
Using Global Transactions 1.5.12.3
Restrictions 1.5.12.4
Developer’s Guide 1.6
Developing JEE Connectors 1.6.1
Connector Environment Setup 1.6.1.1
Build Environment 1.6.1.1.1
Archetype Template Connector Project 1.6.1.1.2
Implementing the Teiid Framework 1.6.1.2
ra.xml file Template 1.6.1.2.1
Packaging the Adapter 1.6.1.3
Adding Dependent Libraries 1.6.1.3.1
Deploying the Adapter 1.6.1.4
Translator Development 1.6.2
Environment Setup 1.6.2.1
Setting up the build environment 1.6.2.1.1
Archetype Template Translator Project 1.6.2.1.2
Implementing the Framework 1.6.2.2
Caching API 1.6.2.2.1
Command Language 1.6.2.2.2
Connections to Source 1.6.2.2.3
Dependent Join Pushdown 1.6.2.2.4

Executing Commands 1.6.2.2.5
Extending the ExecutionFactory Class 1.6.2.2.6
Large Objects 1.6.2.2.7
Translator Capabilities 1.6.2.2.8
Translator Properties 1.6.2.2.9
Extending The JDBC Translator 1.6.2.3
Delegating Translator 1.6.2.4
Packaging 1.6.2.5
Adding Dependent Modules 1.6.2.5.1
Deployment 1.6.2.6
User Defined Functions 1.6.3
Source Supported Functions 1.6.3.1
Support for User-Defined Functions(Non-Pushdown) 1.6.3.2
Archethype Template UDF Project 1.6.3.2.1
AdminAPI 1.6.4
Custom Logging 1.6.5
Runtime Updates 1.6.6
Custom Metadata Repository 1.6.7
PreParser 1.6.8
Archethype Template PreParser Project 1.6.8.1
Embedded Guide 1.7
Logging in Teiid Embedded 1.7.1
Secure Embedded with PicketBox 1.7.2
Reference Guide 1.8
Data Sources 1.8.1
Virtual Databases 1.8.2
Developing a Virtual Database 1.8.2.1
DDL VDB 1.8.2.2
Using XML & DDL 1.8.2.3
VDB Properties 1.8.2.4
Schema Object DDL 1.8.2.5
MultiSource Models 1.8.2.6
Metadata Repositories 1.8.2.7
REST Service Through VDB 1.8.2.8
SQL Support 1.8.3
Identifiers 1.8.3.1
Expressions 1.8.3.2
Criteria 1.8.3.3
Scalar Functions 1.8.3.4
Numeric Functions 1.8.3.4.1

String Functions 1.8.3.4.2
Date_Time Functions 1.8.3.4.3
Type Conversion Functions 1.8.3.4.4
Choice Functions 1.8.3.4.5
Decode Functions 1.8.3.4.6
Lookup Function 1.8.3.4.7
System Functions 1.8.3.4.8
XML Functions 1.8.3.4.9
JSON Functions 1.8.3.4.10
Security Functions 1.8.3.4.11
Spatial Functions 1.8.3.4.12
Miscellaneous Functions 1.8.3.4.13
Nondeterministic Function Handling 1.8.3.4.14
DML Commands 1.8.3.5
Set Operations 1.8.3.5.1
Subqueries 1.8.3.5.2
WITH Clause 1.8.3.5.3
SELECT Clause 1.8.3.5.4
FROM Clause 1.8.3.5.5
XMLTABLE 1.8.3.5.5.1
ARRAYTABLE 1.8.3.5.5.2
OBJECTTABLE 1.8.3.5.5.3
TEXTTABLE 1.8.3.5.5.4
WHERE Clause 1.8.3.5.6
GROUP BY Clause 1.8.3.5.7
HAVING Clause 1.8.3.5.8
ORDER BY Clause 1.8.3.5.9
LIMIT Clause 1.8.3.5.10
INTO Clause 1.8.3.5.11
OPTION Clause 1.8.3.5.12
DDL Commands 1.8.3.6
Temp Tables 1.8.3.6.1
Alter View 1.8.3.6.2
Alter Procedure 1.8.3.6.3
Alter Trigger 1.8.3.6.4
XML SELECT Command 1.8.3.7
Query Structure 1.8.3.7.1
Document Generation 1.8.3.7.2
Procedures 1.8.3.8
Procedure Language 1.8.3.8.1

Virtual Procedures 1.8.3.8.2
Update Procedures 1.8.3.8.3
Comments 1.8.3.9
Datatypes 1.8.4
Supported Types 1.8.4.1
Type Conversions 1.8.4.2
Special Conversion Cases 1.8.4.3
Escaped Literal Syntax 1.8.4.4
Updatable Views 1.8.5
preserved Table 1.8.5.1
Transaction Support 1.8.6
AutoCommitTxn Execution Property 1.8.6.1
Updating Model Count 1.8.6.2
JDBC and Transactions 1.8.6.3
Transactional Behavior with JBoss Data Source Types 1.8.6.4
Limitations and Workarounds 1.8.6.5
Data Roles 1.8.7
Permissions 1.8.7.1
Role Mapping 1.8.7.2
XML Definition 1.8.7.3
Customizing 1.8.7.4
System Schema 1.8.8
SYS 1.8.8.1
SYSADMIN 1.8.8.2
Translators 1.8.9
Amazon SimpleDB Translator 1.8.9.1
Apache Accumulo Translator 1.8.9.2
Apache SOLR Translator 1.8.9.3
Cassandra Translator 1.8.9.4
Delegating Translators 1.8.9.5
File Translator 1.8.9.6
Google Spreadsheet Translator 1.8.9.7
Infinispan HotRod Translator 1.8.9.8
Infinispan Library Mode Translator 1.8.9.9
JDBC Translators 1.8.9.10
Actian Vector Translator 1.8.9.10.1
Apache Phoenix Translator 1.8.9.10.2
Cloudera Impala Translator 1.8.9.10.3
DB2 Translator 1.8.9.10.4
Derby Translator 1.8.9.10.5

Greenplum Translator 1.8.9.10.6
H2 Translator 1.8.9.10.7
Hive Translator 1.8.9.10.8
HSQL Translator 1.8.9.10.9
Informix Translator 1.8.9.10.10
Ingres Translators 1.8.9.10.11
Intersystems Cache Translator 1.8.9.10.12
JDBC ANSI Translator 1.8.9.10.13
JDBC Simple Translator 1.8.9.10.14
MetaMatrix Translator 1.8.9.10.15
Microsoft Access Translators 1.8.9.10.16
Microsoft SQL Server Translator 1.8.9.10.17
ModeShape Translator 1.8.9.10.18
MySQL Translators 1.8.9.10.19
Netezza Translator 1.8.9.10.20
Oracle Translator 1.8.9.10.21
OSISoft PI Translator 1.8.9.10.22
PostgreSQL Translator 1.8.9.10.23
PrestoDB Translator 1.8.9.10.24
Redshift Translator 1.8.9.10.25
SAP Hana Translator 1.8.9.10.26
SybaseIQ Translator 1.8.9.10.27
Sybase Translator 1.8.9.10.28
Teiid Translator 1.8.9.10.29
Teradata Translator 1.8.9.10.30
Vertica Translator 1.8.9.10.31
JPA Translator 1.8.9.11
LDAP Translator 1.8.9.12
Loopback Translator 1.8.9.13
Microsoft Excel Translator 1.8.9.14
MongoDB Translator 1.8.9.15
Object Translator 1.8.9.16
OData Translator 1.8.9.17
OData V4 Translator 1.8.9.18
Swagger Translator 1.8.9.19
OLAP Translator 1.8.9.20
Salesforce Translators 1.8.9.21
SAP Gateway Translator 1.8.9.22
Web Services Translator 1.8.9.23
Federated Planning 1.8.10

Planning Overview 1.8.10.1
Query Planner 1.8.10.2
Query Plans 1.8.10.3
Federated Optimizations 1.8.10.4
Subquery Optimization 1.8.10.5
XQuery Optimization 1.8.10.6
Federated Failure Modes 1.8.10.7
Conformed Tables 1.8.10.8
Architecture 1.8.11
Terminology 1.8.11.1
Data Management 1.8.11.2
Query Termination 1.8.11.3
Processing 1.8.11.4
BNF for SQL Grammar 1.8.12
Security Guide 1.9
LoginModules 1.9.1
Teiid Server Transport Security 1.9.2
JDBC/ODBC SSL connection using self-signed SSL certificates 1.9.3
Data Source Security 1.9.4
Kerberos support through GSSAPI 1.9.5
Custom Authorization Validator 1.9.6
SAML Based Security For OData 1.9.7
OAuth2 Based Security For OData Using KeyCloak 1.9.8
SAML Based Security For OData Using KeyCloak 1.9.9

Legal Notice

Contribute to Teiid Documentation

The pages themselves have commenting enabled. You should be able to click on any text area to add a comment.

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

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 you finished this steps, the content of documents will updated automatically.

Test locally

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

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

Once above commands executes successfully, the http format document can be test 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.

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

Copyright © 2016 by Red Hat, Inc. This copyrighted material is made available to anyone wishing to use, modify, copy, or redistribute it subject to the terms and conditions of the GNU Lesser General Public License, as published by the Free Software Foundation.

Red Hat and the Red Hat "Shadow Man" logo are registered trademarks of Red Hat, Inc. in the United States and other countries.

All other trademarks referenced herein are the property of their respective owners.

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 Teiid instance. This guide guides user through installation of Teiid Server, 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 here http://teiid.jboss.org/basics/

Installation Guide

Teiid needs to be installed into an existing WildFly 10.0.0 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 9.2 directory structure matches WildFly directly - it is just an overlay. This will add necessary modules and configuration files to install Teiid in WildFly 10.0.0 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 10.0.0 , 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 10.0.0 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 10.0.0 and Teiid 9.2 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 10.0.0 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=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 9.2 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 10.0.0 system
/modules/system/layers/base/org/jboss/teiid/client	This directory contains Teiid client libraries. It has the Teiid JDBC driver jar, "teiid-9.2.0.Final-jdbc.jar", and also contains "teiid-hibernate-dialect-9.2.0.Final.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.

Deploying VDBs

A VDB is the primary means to define a Virtual Database in Teiid. A user can create a VDB using Teiid Designer - http://www.jboss.org/teiiddesigner/ - or follow the instructions in the Reference Guide to create a VDB without Teiid Designer.

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 10.0.0 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.

AdminShell Deployment

Teiid provides a groovy based AdminShell scripting tool, which can be used to deploy a VDB. See the "deploy" method. Consult the AdminShell documentation for more information. Note that using the AdminShell scripting, you can automate deployment of artifacts in your environment. When using AdminShell, 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 AdminShell 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 10.0.0 during installation. There are many ways to create a Accumulo data source, using CLI, AdminShell, 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:/accumul
o-ds, class-name=org.teiid.resource.adapter.accumulo.AccumuloManagedConnectionFactory, enabled=true, use-java-c
ontext=true)
/subsystem=resource-adapters/resource-adapter=accumulo/connection-definitions=teiid/config-properties=ZooKeeper
ServerList: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=InstanceN
ame: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 10.0.0 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 10.0.0 during installation. There are many ways to create a SimpleDB data source, using CLI, AdminShell, 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:/si
mpledbDS, class-name=org.teiid.resource.adapter.simpledb.SimpleDBManagedConnectionFactory, enabled=true, use-ja
va-context=true)
/subsystem=resource-adapters/resource-adapter=simpledb/connection-definitions=simpledbDS/config-properties=Acce
ssKey:add(value=xxx)
/subsystem=resource-adapters/resource-adapter=simpledb/connection-definitions=simpledbDS/config-properties=Secr
etAccessKey: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 10.0.0 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 10.0.0 during installation. There are many ways to create a Cassandra data source, using CLI, AdminShell, 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 10.0.0 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.

File Data Sources

File data sources use a Teiid specific JCA connector that is deployed into WildFly 10.0.0 during installation. There are many ways to create the file data source, using CLI, AdminShell, 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, cl
ass-name=org.teiid.resource.adapter.file.FileManagedConnectionFactory, enabled=true, use-java-context=true)
/subsystem=resource-adapters/resource-adapter=file/connection-definitions=fileDS/config-properties=Parentdirect
ory:add(value=/home/rareddy/testing/)
/subsystem=resource-adapters/resource-adapter=file/connection-definitions=fileDS/config-properties=AllowParentP
aths: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 10.0.0 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 10.0.0 during installation. There are many ways to create the Ftp/Ftps data source, using CLI, AdminShell, 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 10.0.0 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

Authentication to your google account may be done in two ways. ClientLogin (requires login and password) or OAuth (more complicated, requires user to supply refresh token).

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.

Config property	Description
AuthMethod	Method to access Google. This property can only be OAuth2.
RefreshToken	Use guide below to retrieve RefreshToken. Request access to Google Drive and Spreadsheet API.
SpreadsheetName	Required property with name of the Spreadsheet that is datasource for this connector.
BatchSize	Maximum number of rows that can be fetched at a time. Defaults to 4096.

Getting an OAuth Refresh Token

Use this request:

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=<AUTH_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
```

The refresh token will be in the response

Implementation Details

Google Translator is implemented using GData API [1] (java library for [4]) and Google Visualization Protocol [2]. Authentication is implemented using OAuth2 for devices [3]. 1. https://developers.google.com/google-apps/spreadsheets/ 2. https://developers.google.com/chart/interactive/docs/querylanguage 3. https://developers.google.com/accounts/docs/OAuth2ForDevices 4. https://developers.google.com/google-apps/spreadsheets

JBoss Data Grid (JDG) Library Mode Data Sources

JDG Libary Mode data sources use a Teiid specific JCA connector that is deployed into WildFly 10.0.0 during installation. This connector has the following options that can be configured to access an JDG cache running in library mode:

- Using JNDI
- Using configuration file

It is assumed that you will be installing the **JDG Library Mode EAP Modules Kit**, which can be obtained on the Red Hat's Customer Portal at https://access.redhat.com/jbossnetwork/restricted/listSoftware.html.

Configuration

Accessing Cache in Library Mode

One of the following properties is required for defining how the cache will be accessed via the CacheManager:

Property Name	Description
CacheJndiName	JNDI name used to find the CacheContainer
ConfigurationFileNameForLocalCache	The JDG Configuration xml file will be used to create and configure the CacheManager as a local cache

Configuring Which Cache and Pojo

The following property is required as it provides the mapping to the JDG cache and pojo that will be accessed.

Property Name	Property Template	Description
CacheTypeMap	cacheName:className[:pkFieldName[:cacheKeyJavaType]]	Defines the cache name and pojo components for accessing the cache

Where

- cacheName - Name of the cache to access
- className - Class name of the pojo that is used to store the data in the cache
- pkFieldName - [Optional] identifies the pojo class attribute that is used as the primary key. It is required for updates or if the pojo contains complex objects used in relationships (1-to-1 or 1-to-many). The pkFieldName **MUST** match a corresponding getter/setter method in the pojo
- cacheKeyJavaType - [Optional] identify primary key java type when different than class attribute type

The following configures the module that contains the pojo stored in the cache:

Property Name	Req.	Description
module	N	Specify the WildFly module that contains the cache classes that were defined in CacheTypeMap

Pojo Jar

The pojo class is the object that will be used to store the data in the cache. It should be built accordingly:

- To take advantage of the cache being indexed enabled, should annotate the class. See [JDG documentation|https://access.redhat.com/documentation/en-US/Red_Hat_JBoss_Data_Grid/6.6/html-single/Infinispan_Query_Guide/index.html#chap-Annotating_Objects_and_Querying]
- The class should be packaged into a jar so that it can be deployed as a module

To configure the use of the pojo, do the following:

- Deploy the pojo jar as a module in the jboss-as server. Then define the "lib" property in the -vdb.xml and assign the correct module name. This can be done using the following template:

```
      <property name ="lib" value ="{pojo_module_name}"></property>
```

- The JDG commons module, org.infinispan.commons, slot="jdg-6.x" or slot for version installed, needs to have the pojo dependency added:

```
      <module name="{pojo_module_name}"   export="true" />
```

Using Cache for External Materialization

The following are the additional properties that need to be configured if using the Remote Cache for external materialization:

Property Name	Req.	Description
StagingCacheName	Y	Cache name for the staging cache used in materialization
AliasCacheName	Y	Cache name for the alias cache used in tracking aliasing of the caches used in materialization

Server Configuration

There are many ways to create the data source; using CLI, AdminShell, 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 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=infinispan/connection-definitions=infinispanDS:add(jndi-name=java
:/infinispanDS, class-name=org.teiid.resource.adapter.infinispan.libmode.InfinispanManagedConnectionFactory, en
abled=true, use-java-context=true)
/subsystem=resource-adapters/resource-adapter=infinispan/connection-definitions=infinispanDS/config-properties=
CacheTypeMap:add(value=trades:org.somewhere.Trade;tradeId)
/subsystem=resource-adapters/resource-adapter=infinispan/connection-definitions=infinispanDS/config-properties=
Module:add(value=org.somewhere)
/subsystem=resource-adapters/resource-adapter=infinispan/connection-definitions=infinispanDS/config-properties=
CacheJndiName:add(value=java:/myCache)
runbatch
```

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 10.0.0 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.

Note	**Avoid Classloading Issues:** If you are using a servlet or other type of web application to create the DefaultCacheManager for the cache, be sure not to include the Infinispan jar dependencies in the application, but add their module dependencies.

JBoss Data Grid (JDG) HotRod Data Sources

JDG caches running in client/server mode can use a Teiid specific JCA connector for accessing as a data source, which is deployed into WildFly 10.0.0 during installation. This connector can be configured to support the accessing of a remote JDG cache using the the Hot Rod client.

It is assumed that you will be installing the **JDG Hotrod Client EAP Modules Kit**, which can be obtained on the Red Hat's Customer Portal at https://access.redhat.com/jbossnetwork/restricted/listSoftware.html.

To handle pojo types in the same cache, where each pojo object can be accessed by performing a "get(key)" on the cache, will require a different configured resource-adapter for each pojo type.

There are two options for how the JDG schema can be configured in the connector; protobuf annotations or protobuf (.proto) file with marshaller(s). The following are the requirements.

Requirement

- (option 1) Minimum, JDG 6.2 - this requires you provide a protobuf definition file and pojo marshaller(s) for the pojo to configure the JDG schema
- (option 2) Minimum, JDG 6.6 - this can be used when the pojo has defined protobuf annotations which are used to configure the JDG schema.

See Infinispan HotRod Translator for details on how the the metadata will be exposed or can be manually configured based on the schema that's defined for this data source.

Configuration

There are many different ways to configure this resource-adapter based on different usage scenarios:

- complexity of the pojo (i.e, pojo contains other pojo's).
- using either pojo protobuf annotations or protobuf definition files, to configure the JDG schema
- wanting the enable HotRod client to use SSL
- utilizing the JDG cache for materialization

Accessing HotRod Remote Cache

One of the following properties is required for defining how the RemoteCacheManager will be created/accessed:

Property Name	Req.	Property Template	Description
CacheJndiName	N		JNDI name to find the CacheContainer
RemoteServerList	N	host:port\[;host:port….\]	Specify the host and ports that will be clustered together to access the caches
HotRodClientPropertiesFile	N		The HotRod properties

			cache

Configuring Which Cache and Pojo

The following property is required as it provides the mapping to the JDG cache and pojo that will be accessed.

Property Name	Property Template	Description
CacheTypeMap	cacheName:className[:pkFieldName[:cacheKeyJavaType]]	Defines the cache name and pojo components for accessing the cache

Where

- cacheName - Name of the cache to access
- className - Class name of the pojo that is used to store the data in the cache
- pkFieldName - [Optional] identifies the pojo class attribute that is used as the primary key. It is required for updates or if the pojo contains complex objects used in relationships (1-to-1 or 1-to-many). The pkFieldName **MUST** match a corresponding getter/setter method in the pojo
- cacheKeyJavaType - [Optional] identify primary key java type when different than class attribute type

JDG Schema using Protobuf Definition and Marshaller(s)

The following properties are required when the protobuf definition file (.proto) and the pojo marshaller(s) are being used to configure the JDG schema:

Property Name	Req.	Property Template	Description
ProtobufDefinitionFile	Y		Path to the Google Protobuf descriptor file that’s packaged in a jar (ex: /quickstart/addressbook.proto)
MessageMarshallers	Y	className:marshallerClassName \ [,className:marshallerClassName,..\]	Contains Class name(s) mapped to its respective message marshaller(s) that is to be registered for serialization
MessageDescriptor	Y		Message descriptor package name in the protobuf descriptor file

The following property should be defined when using protobuf definition file and marshallers:

Property Name	Req.	Property Template	Description
module	N		Specify the WildFly module that contains the cache classes that need to be loaded

JDG Schema using Pojo Annotations

The pojo class is the object that is used to store the data in the cache. It should be built accordingly:

- If the pojo is to be used to define the schema, then should use the protobuf annotations. See JDG Protobuf Annotations at https://access.redhat.com/documentation/en-US/Red_Hat_JBoss_Data_Grid/6.6/html-single/Infinispan_Query_Guide/index.html#Custom_Fields_Indexing_with_Protobuf
- The class should be packaged into a jar so that it can be deployed as a module.
- If the root pojo, defined in CacheTypeMap property, contains other pojo(s) (e.g., 1-to-1 or 1-to-many relationship), then those child pojo's must be defined in the ChildClasses property so that they can get registered in the JDG schema. The following property must be configured:

Property Name	Property Template	Description
ChildClasses	className[,className,..]]	comma separated list of class names that indicate the child classes to register in the JDG schema

Enable Hotrod Client SSL

To enable the Hotrod client in the resource-adapter to communicate using SSL, configure the following properties on the resource adapter:

Property Name	Description
TrustStoreFileName	File name of the truststore
TrustStorePassword	TrustStore Password
KeyStoreFileName	File name of the keystore
KeyStorePassword	KeyStore Password
SNIHostName	[optional] SNI Host Name

Using for External Materialization

The following are the additional properties that need to be configured if using the Remote Cache for external materialization:

Property Name	Req.	Description
StagingCacheName	Y	Cache name for the staging cache used in materialization
AliasCacheName	Y	Cache name for the alias cache used in tracking aliasing of the caches used in materialization. This cache can be shared with other configured materializations.

Examples

There are many ways to create the data source, using CLI, AdminShell, admin-console etc.

1st Example

The 1st example is an xml snippet of a resource-adapter that is using probufs and marshallers to configure the JDG schema:

Sample Resource Adapter defining Protobuf Definition and Marshaller

```
               <resource-adapter id="infinispanRemQS">
                    <module slot="main" id="org.jboss.teiid.resource-adapter.infinispan.hotrod"/>
                    <connection-definitions>
                        <connection-definition class-name="org.teiid.resource.adapter.infinispan.hotrod.Infinis
panManagedConnectionFactory" jndi-name="java:/infinispanRemote" enabled="true" use-java-context="true" pool-name
="infinispanDS">
                            <config-property name="CacheTypeMap">
                                addressbook:org.jboss.as.quickstarts.datagrid.hotrod.query.domain.Person;id
                            </config-property>
                            <config-property name="ProtobufDefinitionFile">
                                 /quickstart/addressbook.proto
                            </config-property>
                            <config-property name="MessageDescriptor">
                                quickstart.Person
                            </config-property>
                            <config-property name="Module">
                                com.client.quickstart.pojos
                            </config-property>
                            <config-property name="MessageMarshallers">                                org.jboss.
as.quickstarts.datagrid.hotrod.query.domain.Person:org.jboss.as.quickstarts.datagrid.hotrod.query.marshallers.P
ersonMarshaller,org.jboss.as.quickstarts.datagrid.hotrod.query.domain.PhoneNumber:org.jboss.as.quickstarts.data
grid.hotrod.query.marshallers.PhoneNumberMarshaller,org.jboss.as.quickstarts.datagrid.hotrod.query.domain.Phone
Type:org.jboss.as.quickstarts.datagrid.hotrod.query.marshallers.PhoneTypeMarshaller
                            </config-property>
                            <config-property name="RemoteServerList">
                                127.0.0.1:11322
                            </config-property>
                        </connection-definition>
                    </connection-definitions>
                </resource-adapter>
```

2nd Example

The 2nd example showing a pojo example with annotations and the xml snippet of the resource-adapter:

Sample Pojo with Annotations

```
public class Person {

   @ProtoField(number = 2, required = true)
   public String name;
   @ProtoField(number = 1, required = true)
   public int id;
   @ProtoField(number = 3)
   public String email;
   private List<PhoneNumber> phones;

   public String getName() {
      return name;
   }

   public void setName(String name) {
      this.name = name;
   }

   public int getId() {
      return id;
   }

   public void setId(int id) {
      this.id = id;
   }
```

```
    public String getEmail() {
       return email;
    }

    public void setEmail(String email) {
       this.email = email;
    }

    public List<PhoneNumber> getPhones() {
       return phones;
    }

    public void setPhones(List<PhoneNumber> phones) {
       this.phones = phones;
    }
}
```

Sample Resource Adapter using Pojo with annotations

```
               <resource-adapter id="infinispanRemQSDSL">
                    <module slot="main" id="org.jboss.teiid.resource-adapter.infinispan.dsl"/>
                    <connection-definitions>
                        <connection-definition class-name="org.teiid.resource.adapter.infinispan.dsl.Infinispan
ManagedConnectionFactory" jndi-name="java:/infinispanRemoteDSL" enabled="true" use-java-context="true" pool-name
="infinispanRemoteDSL">
                            <config-property name="RemoteServerList">
                                127.0.0.1:11322
                            </config-property>
                            <config-property name="CacheTypeMap">
                                addressbook_indexed:org.jboss.as.quickstarts.datagrid.hotrod.query.domain.Perso
n;id
                            </config-property>
                            <config-property name="ChildClasses">
                               org.jboss.as.quickstarts.datagrid.hotrod.query.domain.PhoneNumber
                            </config-property>
                        </connection-definition>
                    </connection-definitions>
                </resource-adapter>
```

3rd Example

The 3rd example is using the JDG data source for materialization.

Sample Resource Adapter for external materialization

```
                <resource-adapter id="infinispanRemQSDSL">
                    <module slot="main" id="org.jboss.teiid.resource-adapter.infinispan.hotrod"/>
                    <connection-definitions>
                        <connection-definition class-name="org.teiid.resource.adapter.infinispan.hotrod.Infinis
panManagedConnectionFactory" jndi-name="java:/infinispanRemoteDSL" enabled="true" use-java-context="true" pool-
name="infinispanRemoteDSL">
                            <config-property name="CacheTypeMap">
                                addressbook_indexed:org.jboss.as.quickstarts.datagrid.hotrod.query.domain.Perso
n;id
                            </config-property>
                            <config-property name="StagingCacheName">
                                addressbook_indexed_mat
                            </config-property>
                            <config-property name="AliasCacheName">
                                aliasCache
                            </config-property>
                            <config-property name="Module">
                                com.client.quickstart.addressbook.pojos
                            </config-property>
```

```
                            <config-property name="RemoteServerList">
                                127.0.0.1:11322
                            </config-property>
                        </connection-definition>
                    </connection-definitions>
                </resource-adapter>
```

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 10.0.0 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, AdminShell, 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 10.0.0 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 10.0.0 during installation. There are many ways to create the ldap data source, using CLI, AdminShell, 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 10.0.0 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 LdapAdminUserDN and LdapAdminUserPassword to empty values.

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 10.0.0 during installation. There are many ways to create a MongoDB data source, using CLI, AdminShell, 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:/mongoDS", class-name=org.teiid.resource.adapter.mongodb.MongoDBManagedConnectionFactory, enabled=true, use-java-context=true)
/subsystem=resource-adapters/resource-adapter=mongodb/connection-definitions=mongodbDS/config-properties=RemoteServerList:add(value="{host}:27017")
/subsystem=resource-adapters/resource-adapter=mongodb/connection-definitions=mongodbDS/config-properties=Database: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		
Username	Connection User’s Name	false	none
Password	Connection User’s password	false	none
Database	MongoDB database name	true	none
SecurityType	MongoDB Type of Authentication to be used. Allowed values are "None","SCRAM_SHA_1","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)
```

Developer’s Tip - If WildFly 10.0.0 is running in standalone mode, you can also manually edit the "<jboss-

Tip	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.

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.api,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-url=jdbc:phoenix:{zookeeper quorum server}, enabled=true, use-java-context=true, user-name={user}, password={password})
/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.

Salesforce Data Sources

Salesforce data sources use a Teiid specific JCA connector that is deployed into WildFly 10.0.0 during installation. There are two versions of the salesforce resource adapter - salesforce, which currently provides connectivity to the 22.0 Salesforce API and salesforce-34, which provides connectivity to the 34.0 Salesforce API. The version 22.0 support has been deprecated.

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.

There are many ways to create the salesforce data source, using CLI,AdminShell, 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://www.salesforce.com/services/Soap/u/22.0)
/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-34 connection definition configuration is similar to the above. The resource adapter name would instead be salesforce-34, and the url would point to 34.0 instead.

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 10.0.0 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=ConfigFi
le: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/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>
```

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

h== 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

Solr Data Sources

Solr data sources use a Teiid specific JCA connector that is deployed into WildFly 10.0.0 during installation. There are many ways to create a Solr data source, using CLI, AdminShell, 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, cl
ass-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(valu
e=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 10.0.0 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 10.0.0 during installation. There are many ways to create the file data source, using CLI, AdminShell, 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 10.0.0 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	true	n/a	URL for HTTP, Service Endpoint for SOAP
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 & 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 WS-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[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

The CXF config property may also be used to control the logging of requests and responses for specific or all ports. Logging, when enabled, will be performed at an INFO level to the org.apache.cxf.interceptor context.

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>
```

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>
```

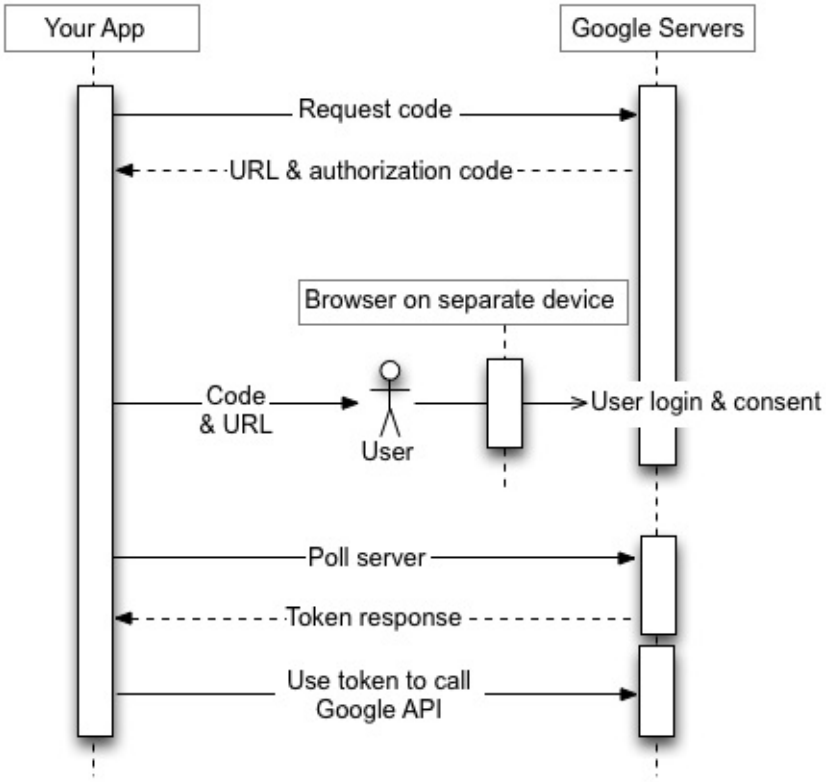
If in case there is no delegationCredential is available on the context, the access will fail.

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.xml, which is useful for an xml file deployment, 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.

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 aynch appender to minimize the performance impact. For example you can use a configuration snippet like the one below to insert an asynch 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. See the container's *standalone-teiid.xml* for a more complete listing of logging contexts used in the container.

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	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 plan commands (for when the query planning is performed 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.

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 number of pairs depending upon your query.

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.

Note	**"Want to log to a database?"** - If you would like to log Command log messages to any database, then look at the https://github.com/teiid/teiid-extensions project. The installation zip file is available in Teiid downloads page.

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.

Note	**"Want to log to a database?"** - If you would like to log Audit log messages to any database, then look at the https://github.com/teiid/teiid-extensions project. The installation zip file will be available in Teiid downloads page.

To enable audit logging to an alternative file location, configure a separate file appender for the DETAIL logging of the org.teiid.AUDIT_LOG context. See the Developer's Guide to develop a custom logging solution if file based, or any other built-in Log4j, logging is not sufficient.

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 the Client Developer's Guide on how to use 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 or Admin Shell. Copying VDB directly into the "deployments" directory is not supported.

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, AdminShell 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-service" 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-service-use-disk,value=true)
/subsystem=teiid:write-attribute(name=buffer-service-encrypt-files,value=false)
/subsystem=teiid:write-attribute(name=buffer-service-processor-batch-size,value=256)
/subsystem=teiid:write-attribute(name=buffer-service-max-open-files,value=64)
/subsystem=teiid:write-attribute(name=buffer-service-max-file-size,value=2048)
/subsystem=teiid:write-attribute(name=buffer-service-max-processing-kb,value=-1)
/subsystem=teiid:write-attribute(name=buffer-service-max-reserve-kb,value=-1)
/subsystem=teiid:write-attribute(name=buffer-service-max-buffer-space,value=51200)
/subsystem=teiid:write-attribute(name=buffer-service-max-inline-lobs,value=true)
/subsystem=teiid:write-attribute(name=buffer-service-memory-buffer-space,value=-1)
/subsystem=teiid:write-attribute(name=buffer-service-max-storage-object-size,value=8388608)
/subsystem=teiid:write-attribute(name=buffer-service-memory-buffer-off-heap,value=false)
```

Note	Note that 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.

max-reserve-kb (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 *max-reserve-kb* will at a max use: (1024-300) * 0.4) + (7 * 1024 * 0.5 = 4373.6 MB or 4,478,566 KB

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 *max-reserve-kb* 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.

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 *max-reserve-kb*. Typical minimum memory required by Teiid when all the active plans are active is *#active-plans*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 *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 max-processing-kb on your own. Typically it will not need adjusted unless you are seeing situations where plans seem memory constrained with low performing large sorts.

max-file-size (default 2GB) - 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 *max-file-size* megabytes. Consider increasing the storage space available to all such files by increasing *max-buffer-space*, if your installation makes use of internal materialization, makes heavy use of SQL/XML, or processes large row counts.

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.

max-storage-object-size (default 8288608 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-kb*. *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 max-storage-object-size to support larger storage objects. Alternatively you could make the processor-batch-size smaller.

memory-buffer-space (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 *max-reserve-kb*. The memory buffer supports only 1 concurrent writer for each *max-storage-object-size* of the *memory-buffer-space*. 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 *max-reserve-kb* size. If the memory buffer is on heap and the *max-reserve-kb* is automatically calculated, then the memory buffer space will be subtracted out of the effective *max-reserve-kb*. If the memory buffer is off heap and the *max-reserve-kb* is automatically calculated, then it's size will be reduced slightly to allow for effectively more

	working memory in the vm.

memory-buffer-off-heap (default false) - Take advantage of the *BufferManager* memory buffer to access system memory without allocating it to the heap. Setting *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 **memory-buffer-space** setting) accounting for any additional direct memory that may be needed by the AS or applications running in the AS.

Disk Usage

max-buffer-space (default -1) - 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 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 *max-buffer-space* setting.

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 max-processing-kb and max-reserve-kb 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.

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 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 memory-buffer-off-heap setting or by restricting the max-reserve-kb setting. Reducing the max-processing-kb setting should generally not be necessary, unless there is a need to severly restrict the heap usage beyond the max-reserve-kb 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.

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 Adminshell. 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 ODBC. Typical installations will not need to adjust the default thread and buffer size 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, such as Designer generated REST VDBs.

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. The Teiid kit does not include files for Teiid Console. This is separate download you can download from Teiid Downloads.

Installation

Once you download the Teiid Console, unzip the contents over the WildFly root directory and all the required files will be overlayed correctly to install Teiid Console. 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

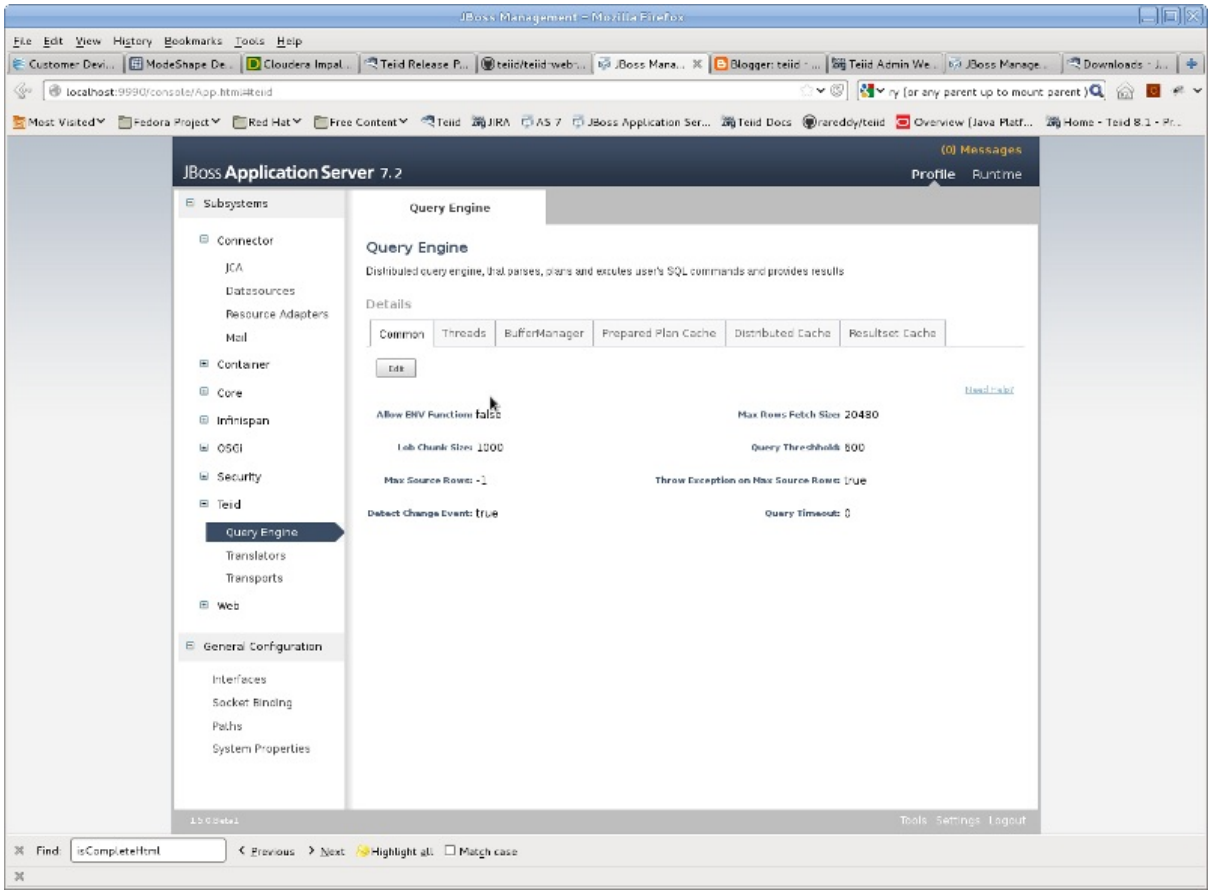
```
add-user.bat
```

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. 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.

Profile View

Profile view is for configuration. Click on "profile" on top right hand corner of the main console screen, to go to Profile View. You can click on "Teiid" in left navigation tree to go to Teiid configuration. There you have three 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.

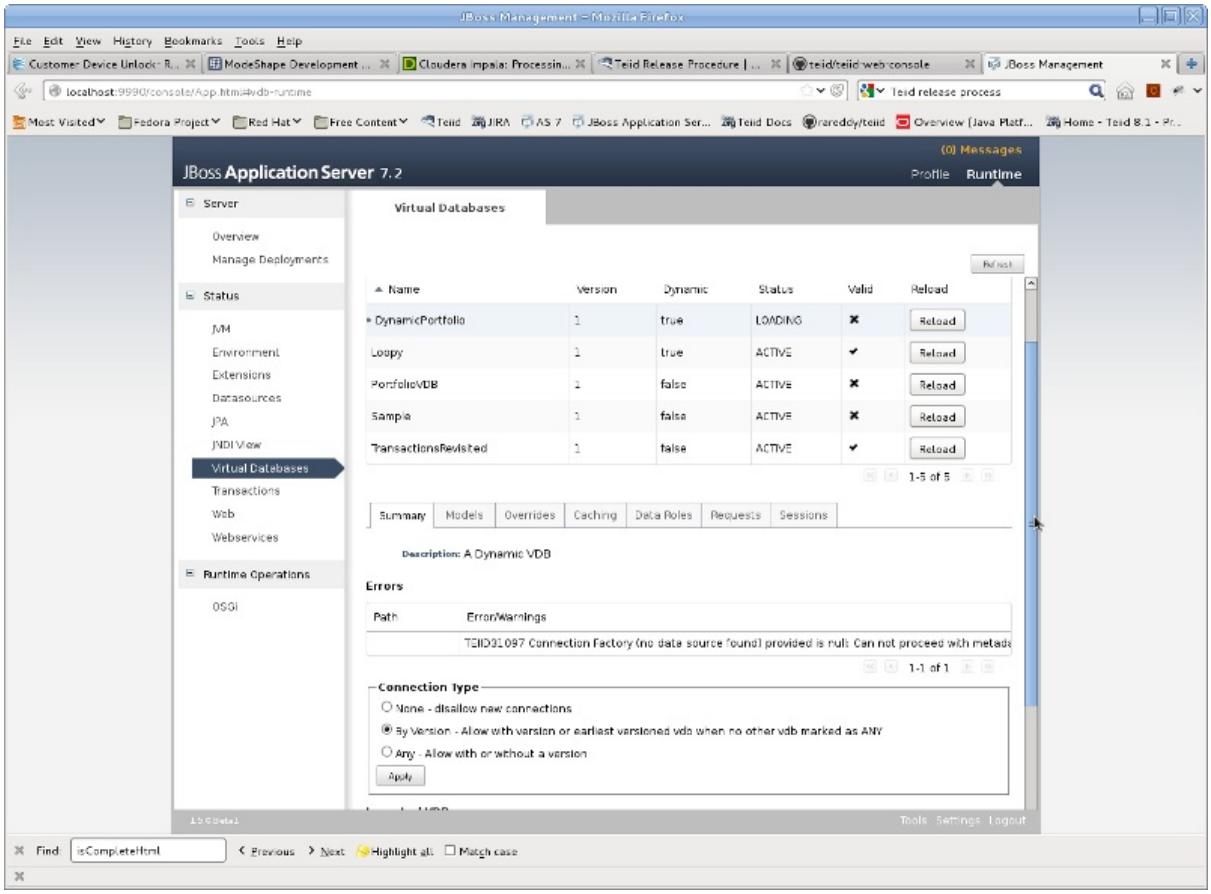


Using this view you can change any configuration value of Teiid. Note that various different configuration properties are sub-divided into different tabs. You can click "Need Help" link on these pages to see the description of each field on the page.

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

Runtime View

Runtime view shows run time information about AS7 and Teiid subsystem. Runtime information about Teiid can be viewed by selecting "Virtual Databases" on left hand navigational tree.



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. Note that this does not work for XML based models that are designed using Teiid Designer.

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 in the Teiid Designer, 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 using the Teiid Designer or hand coded in the vdb.xml file. 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.

AdminShell

The AdminShell tooling provides scripting based programming environments that enable user to access, monitor and control a Teiid Server. Both the command line and graphical console tools are built on functionality provide by the Groovy (http://groovy.codehaus.org/) project. The AdminShell tools can be used in ad-hoc scripting mode or to run pre-defined scripts.

AdminShell features:

1. fully functional programming environment with resource flow control and exception management. See Groovy docs for the full power of the language.
2. quick administrative tool. The user can connect to a running Teiid Server and invoke any of the AdminAPI methods, such as "deploy" or "createDataSource", to control the Teiid System. Since this can be script driven, these tasks can be automated and re-run at a later time.
3. simplified data access tool. The user can connect to a VDB, issue any SQL commands, and view the results of the query via Groovy Sql extensions.
4. migration tool. This can be used to develop scripts like moving the Virtual Databases (VDB), Connection Factories, and Configuration from one development environment to another. This will enable users to test and automate their migration scripts before production deployments.
5. testing tool. The JUnit (http://junit.org) test framework is built in, see Groovy Unit Tests. User can write regression tests for checking system health, or data integrity that can be used to validate a system functionality automatically instead of manual verification by QA personnel.

Download

AdminShell is distributed along with other Teiid downloads under "teiid-{version}-adminshell-dist.zip" name. Download and unzip this file to any directory. Once you have unzipped the file, in root directory you will find "adminshell" and "adminshell-console" executable scripts to launch the command line and graphical tools respectively.

Windows: Double click or execute "adminshell.cmd"

nix: Execute the "adminshell.sh" script

Getting Started

To learn the basics of Groovy take a look at their documents and tutorials on their website.

Basic knowledge of the Java programming language and types is required in order to effectively design and develop scripts using the AdminShell. To learn Java language find learning resources at http://java.sun.com.

You can learn about the Teiid AdminAPI either using "adminHelp()" function or by using the JavaDocs.

AdminShell is a specialized version of Groovy which works in several different modes: interactive shell, graphical console, or script run mode. In interactive shell mode (launched via adminshell), the user can invoke connect to a live Teiid system and issue any ad-hoc commands to control the system. The interactive buffer can be used to develop a script and the interactive session input and output can be captured into a log file, more on this later in the document.

In graphical mode (lanched via adminshell-console), the user can develop and run scripts using a text editor that supports syntax highlighting.

In the script run mode, the user can execute/play back previously developed scripts. This mode especially useful to automate any testing or to perform any repeated configurations/migrations changes to a Teiid system.

Essential Rules

To use AdminShell successfully, there are some basic syntactical rules to keep in mind.

1. In interactive shell mode, most commands (as seen by the help command) are used to control shell behavior and are not general Groovy scripting constructs. Admin methods will typically be called using functional notation:

```
connectAsAdmin()
```

1. All commands and functions are case sensitive.
2. An ending semicolon is optional for Groovy statements.
3. If a function requires input parameter(s), they should be declared inside "(" and ")". A function may have more than one parameter. String parameters can be wrapped in double or single quotes. Example:

```
connectAsAdmin("localhost", "9990", "user", "password", "conn1")
```

1. Local admin connection may not require a password. Remote connections will typically be over the 9993 https port instead. Other Java methods and classes can be used from your scripts, if the required Java class libraries are already in class path. You may place additional jars in the lib directory to have be automatically part of the class path. An example showing an import:

```
import my.package.*;
myObject = new MyClass();
myObject.doSomething();
```

To execute the commands and arbitrary script in interactive mode you enter them first and press enter to execute, then enter the next line, so on.

To exit the tool in the interactive mode, first disconnect if you are connected to the Teiid system by executing "disconnect();" then type "exit". In the script mode, when execution of the script finishes the tool will exit automatically, however you still have to disconnect from Teiid system in the script.

Note	If SSL is turned on the Teiid server, you would need to adjust the connection URL and the client SSL settings as necessary (typically this will only be needed for 2-way SSL).

Help

The adminHelp() methods lists all the available administrative API methods in the AdminShell. Please note that none of the Groovy Shell commands or other available function calls will be shown in this list

```
adminHelp();
```

To get a specific definition about a method and it's required input parameters, use adminHelp("method")

```
adminHelp("deploy");

/*
 *Deploy a VDB from file
 */
void deploy(
    String /* file name */)
  throws AdminException
  throws FileNotFoundException
```

The sqlHelp() methods lists all Sql extension methods.

```
sqlHelp();
```

To get a specific definition about a method and it's required input parameters, use sqlHelp("method")

Basic Commands

The list below contains some common commands used in AdminShell.

Basic Commands

```
println "xxx";   // print something to console

adminHelp();     // shows all the available admin commands;

sql = connect(); // get an extended Groovy Sql connection using connection.properties file

sql.execute(<SQL>); // run any SQL command.

connectAsAdmin(); // connect as admin; no need have the vdb name. SQL commands will not work under this connect
ion

println getConnectionName();  // returns the current connection name

useConnection(<connection name>); // switches to using the given connection settings

disconnect();   // disconnects the current connection in the context
```

Some examples

Example of Deploying a VDB

```
connectAsAdmin();
deploy("/path/to/<name>.vdb");
```

```
// check to validate the deployment
VDB vdb = getVDB("<name>", 1);
if (vdb != null){
     print (vdb.getName()+"."+vdb.getVersion()+" is deployed";
}
else {
     print ("<name>.vdb failed to deploy";
}
```

Create a Datasource(oracle)

```
connectAsAdmin();

// first deploy the JDBC jar file for Oracle
deploy("ojdbc6.jar");

props = new Properties();
props.setProperty("connection-url","jdbc:oracle:thin:@<host>:1521:<sid>");
props.setProperty("user-name", "scott");
props.setProperty("password", "tiger");

createDataSource("oracleDS", "ojdbc6.jar", props);
```

Create a Resource Adapter(file) based Data source

```
connectAsAdmin();

props = new Properties();
props.setProperty("jndi-name","java:/fileDS");
props.setProperty("ParentDirectory", "${jboss.server.base.dir}/myData");
props.setProperty("AllowParentPaths", "true");

// NOTE:  the 2nd argument, template-name, must match the 'id' of one of the resource-adapters that are current
ly defined in the server
createDataSource("MyFile", "file", props);
```

Execute SQL Query against Teiid

```
sql = connect("jdbc:teiid:<vdb>@mm://<host>:31000", "user", "user");

// select
sql.eachRow("select * from sys.tables") { println "${it}" }

// update, insert, delete
sql.execute(<sql command>);
```

Note	Learn more about Groovy SQL

Executing a script file

To execute a script file "foo.groovy" in a directory "some/directory" in the interactive comamnd line tool, execute as following

```
. some/directory/foo.groovy
```

"foo.groovy" is read into current context of the shell as if you typed in the whole document. If your script only contained method calls, you can explicitly invoke the call to execute.

Full execute syntax may also be used, and is required outside of the interactive command line tool:

```
evaluate("some/directory/foo.groovy" as File)
```

To execute the same file without entering interactive mode, run

```
./adminshell.sh . some/directory/foo.groovy
```

Parameters can be passed in as Java System properties. For example

```
./adminshell.sh -Dparam=value . some/directory/foo.groovy
```

Inside the script file, you can access these properties using System.getProperty

```
value = System.getProperty(“param”); // will return "value"
```

Log File and Recorded Script file

During the interactive mode, input is recorded in a history file. This file can be accessed via the up arrow in the interactive shell.

User can also capture the commands entered during a interactive session to their own script file by using "startRecording" and "stopRecording" commands. For example,

```
record start directory/filename.txt
<commands and script ..>
record stop
```

All input and output between the start and stop are captured in the "directory/filename.txt" file. This gives the user an option to capture only certain portions of the interactive session and to later refine a script out of recorded file.

Default Connection Properties

The file "connection.properties" in the installation directory of the AdminShell defines the default connection properties with which user can connect to Teiid system. The following properties can be defined using this file

```
jdbc.user=user
jdbc.password=user
jdbc.url=jdbc:teiid:admin@mm://localhost:31000;

admin.host=localhost
admin.port=9990
admin.user=admin
admin.password=admin
```

A call to "connect()" or "connectionAsAdmin()" without any input parameters, will connect to the Teiid system using the properties defined in properties file. However, a user can always pass in parameters in the connect method to connect to a same or different server than one mentioned in the "connection.properties". Look all the all the different connect methods using the "adminHelp()" method.

Note	Note that it is not secure to leave the passwords in clear text, as defined above. Please take necessary measures to secure the properties file, or do not use this feature and always pass in password interactively or some other secure way.

Note	At any given time user can be actively connected to more than one system or have more than one connection to same system. To manage the connections correctly each connection is created given a unique connection name. To learn more about this look at Handling Multiple Connections.

Handling Multiple Connections

Using AdminShell, a user can actively manage more than one connection to a single or multiple Teiid systems. For example, two separate connections can be maintained, one to the development server and one to the integration server at the same time. This is possible because AdminShell supports a feature called named connections.

Every time a connection is made, the connection has an explicit or an implicitly assigned name. If another connect command is executed then a new connection is made with a unique name and execution will be switched to use the new connection. The previous connection will be held as it is in its current state, and will not be closed.

You can use the following command to find out the current connection's name

```
name = getConnectionName();
```

Knowing the names of the connection that user is working with is important to switch the active connection to a previous connection. To switch the active connection, use the following command and supply the name of the connection to be used

```
useConnection("name");
```

If user supplies the same name as the active connection as they are currently participating in, then this operation will simply return with out any modifications. There is no limitation the number of simultaneous connections.

The following shows an example of using and switching between two connections.

```
// creates a connection
connectAsAdmin();

//capture the connection name
conn1 = getConnectionName();

deploy("file.vdb")

// creates a second connection
connectAsAdmin();

conn2 = getConnectionName();

deploy("file.vdb")

// switch the connection to "conn1"
useConnection(conn1);

// close the connection in the "conn1"
disconnectAll();
```

Interactive Shell Nuances

The interactive shell uses a special shell interpretter and therefore has different behavior than just writting a script in Groovy. See the Groovy Shell Documentation for more on its usage. Notable differences:

- Def statements do not define a variable in the context of the Shell, e.g. do not use `def x = 1`, use `x = 1`
- Shell commands (as seen through `help`) using the non-functional shell syntax are only available in the shell.
- Groovy classes using annotations cannot be parsed in the Shell.

Other Scripting Environments

The AdminShell methods (named contextual connections, AdminAPI wrapper, and help system) have no direct dependencies on Groovy and can be used in other scripting languages.

To use the AdminShell methods in another language, simply import the static methods and Admin classes into your script. You will also need to ensure that the <adminshell dist>/lib/teiid-7.6-client.jar and <adminshell dist>/lib/teiid-adminshell-7.6.jar are in your class path. The snippet below show import statements that would work in Java, BeanShell, Groovy, etc.

```
import static org.teiid.adminshell.AdminShell.*;
import static org.teiid.adminshell.GroovySqlExtensions.*;
import org.teiid.adminapi.*;
```

Note that the provided shell and console executables automatically have the proper class path set and inject the proper imports into running scripts. If you wish to use scripts in a non-Teiid Groovy environment, you will need to manually add these imports and add the admin/client jars to the class path.

System Properties

Some of Teiid's low-level behavior can be configured via system 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`.

General

- *org.teiid.allowNanInfinity* - defaults to false. 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.
- *org.teiid.useValueCache* - defaults to false. 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.
- *org.teiid.ansiQuotedIdentifiers* - defaults to true. 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.
- *org.teiid.subqueryUnnestDefault* - defaults to false. 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.
- *org.teiid.ODBCPacketSize* - defaults to 307200. Target size in bytes of the ODBC results buffer. This is not a hard maximum, lobs and wide rows may use larger buffers.
- *org.teiid.decimalAsDouble* - defaults to false. 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.
- *org.teiid.comparableLobs* - defaults to false. 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.
- *org.teiid.comparableObject* - defaults to false. 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.
- *org.teiid.padSpace* - defaults to false. 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.
- *org.teiid.collationLocale* - not set by default, which means that Java's natural (UTF-16) string comparison will be used. 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.
- *org.teiid.clientVdbLoadTimeoutMillis* - defaults to 5 minutes. 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.
- *org.teiid.enDateNames* - defaults to false. 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.
- *org.teiid.pushdownDefaultNullOrder* - defaults to false. Set to true to mimic 8.1 and prior release behavior of pushing the Teiid's default null order of nulls low if the source has a different default null order and supports explicit null ordering.

- *org.teiid.requireTeiidCollation* - defaults to false. Set to true to force all sorts to be in Teiid's collation (see org.teiid.collationLocale).
- *org.teiid.implicitMultiSourceJoin* - defaults to true. 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.
- *org.teiid.maxStringLength* - defaults to 4000. 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.
- *org.teiid.assumeMatchingCollation* - defaults to false. 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).

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

- *org.teiid.calendarTimestampDiff* - defaults to true. 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.
- *org.teiid.compactBufferFiles* - defaults to false. Set to true to have Teiid keep the buffer files more compact (minimizing sparse regions).
- *org.teiid.maxMessageSize* - defaults to 2097152. 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.
- *org.teiid.maxStreamingLobSize* - defaults to 4294967296. The maximum size of lobs in bytes that are allowed to be streamed as part of the message from clients.
- *org.teiid.defaultIndependentCardinality* - defaults to 10. 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.
- *org.teiid.checkPing* - defaults to true. 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.
- *org.teiid.defaultNullOrder* - defaults to LOW. 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.
- *org.teiid.iso8601Week* - defaults to true. 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.
- *org.teiid.widenComparisonToString* - defaults to false. 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'.
- *org.teiid.aggressiveJoinGrouping* - defaults to true. 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.

Security

- *org.teiid.allowCreateTemporaryTablesByDefault* - defaults to false. Set to true to use the pre-8.0 behavior of allowing any authenticated user to create temp tables without an explicit permission.

- *org.teiid.allowFunctionCallsByDefault* - defaults to false. 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.
- *org.teiid.ignoreUnauthorizedAsterisk* - defaults to false. 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.
- *org.teiid.sanitizeMessages* - defaults to false. 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 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.
- *org.teiid.ODBCRequireSecure* - defaults to true. 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).

PostgreSQL Compatibility

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

- *org.teiid.backslashDefaultMatchEscape* - defaults to false. 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.
- *org.teiid.addPGMetadata* - defaults to true. When set to false, the VDB will not include Postgresql based system metadata.
- *org.teiid.honorDeclareFetchTxn* - defaults to false. When false the wrapping begin/commit of a UseDeclareFetch cursor will be ignored as Teiid does not require a transaction.

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. AdminShell provides a binding into the Groovy scripting language and higher level methods that are often needed when interacting with Teiid. It is still 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
- Authentication Operations
- Source Operations
- Translator Operations
- Runtime Operations

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)
```

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, translator-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.

General Diagnostic Process

- When there is an issue, start by isolating a problem query as much as possible. If accessing through ODBC or OData, does a similar issue occur just through JDBC. 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.

Designer is typically the first place to look at query plans from the query execution view. It’s easiest to look at using Designer or an XML aware viewer as the plan trees can get large.

You can also use Teiid Extensions or SQL statements to get the plan:

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

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 8.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. For better performance the BufferManager setting should be adjusted to the 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 including XML document model 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 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.xml:

```
<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 AdminShell

```
connectAsAdmin()
clearCache("QUERY_SERVICE_RESULT_SET_CACHE")
```

See the Administrator's Guide for more on using the AdminAPI and AdminShell.

Limitations

- XML, BLOB, CLOB, 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. You may also scope the materialized view to be session specific, but that may limit the reuse of the results in many situations.

External Materialization

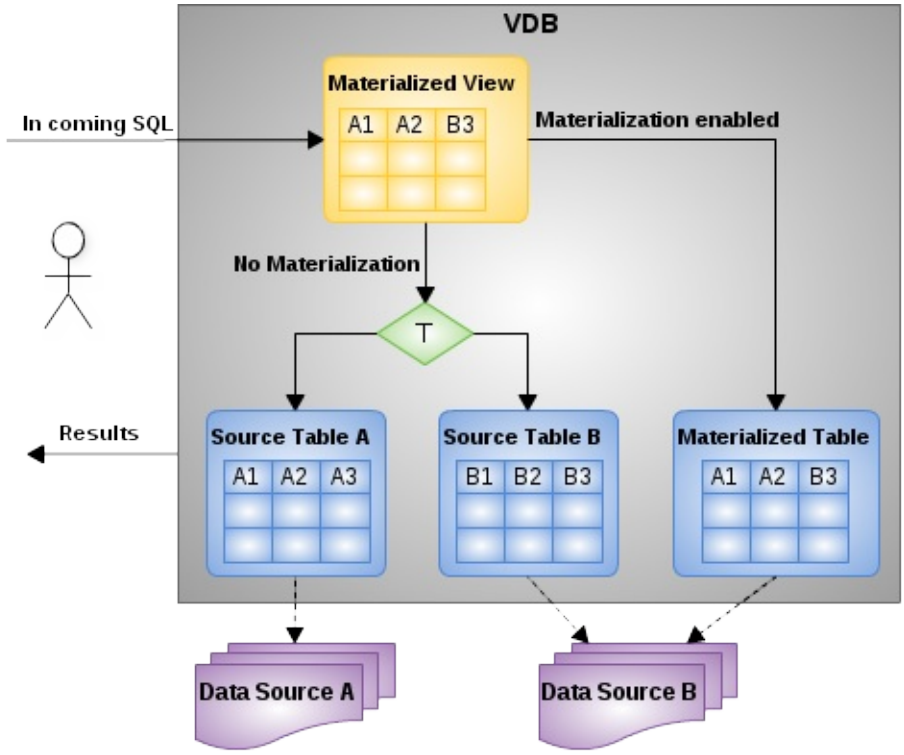
This document will explain what Teiid External Materialization is and how to use it.

Table of Contents

- What is it
- External Materialized Data Source Systems
  - RDBMS Systems
  - JBoss Data Grid (JDG)
- View Options
- Materialization Management
  - 1. Creation of Status Table
  - 2. Creation of View and Materialized Table
  - Materialization Table Loading
  - Eager Materialization Table Update
- Using JDG as Materialization Store
- Define Materialized View in Designer
- Materialization with Embedded Teiid
- Appendix-1: DDL for creating MatView Status Table
- Appendix-2: Example VDB with External Materialized View Options

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.

JBoss Data Grid (JDG)

- JBoss Data Grid (JDG) - for in-memory caching of results.

View Options

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

Property Name	Description	Optional	Default
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 proeprty true, invokes internal materialization.	false	n/a
UPDATABLE	Allow updating Materialized View via DML `UPDATE`	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 View. See below for table structure and DDL for it.	false	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	NONE
teiid_rel:MATVIEW_BEFORE_LOAD_SCRIPT	DDL/DML command to run before the actual load of the cache	true	When not defined, n script will be 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	When not defined, n script will be 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	IMPORTE
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**	true	n/a

	another version of the VDB is still active. Deletion of this information will reload the materialization table.		
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	WAIT
teiid_rel:MATVIEW_TTL	time to live in milliseconds. Provide property or cache hint on view transformation - property takes precedence.	true	2^63 millisecon - effective the table will not refresh, bu will be loaded a single tim initially
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

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

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,
  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).

2. Creation of View and Materialized Table

Define the View and its transformation either using the Designer or directly in DDL in a VDB's model. 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.

Eager Materialization Table Update

SYSADMIN.updateMatView used to perform a eager incremental update based on 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.

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.

Using JDG as Materialization Store

The JDG can be used as a materialization target to save cache contents of the View in a JDG server. JDG can be cluster aware and is accessed using the JDG Hot Rod Client. See the JDG Hot Rod DataSource for how to configure accessing the remote cache.

To configure for external materialization, see the HotRod Translator. This explains how to configure Materialization Management that is specific for using JDG remote cache and is essential for managing the underlying multiple caches needed in order to perform materialization.

Once the Materialized View is completely defined, deploy it to a Teiid Server. Then create a new session and issue a query against Materialized View. You will find it's time-saving and cpu-saving if your query sentences are complex and across multiple, heterogeneous data stores.

Define Materialized View in Designer

- Create materialized views and corresponding physical materialized target tables in Designer. This can be done through setting the materialized and target table manually, or by selecting the desired views, right clicking, then selecting Modeling → "Create Materialized Views"
- Generate the DDL for your physical model materialization target tables. This can be done by selecting the model, right clicking, then choosing Export → "Metadata Modeling" → "Data Definition Language (DDL) File". This script can be used to create the desired schema for your materialization target on whatever source you choose.

Materialization with Embedded Teiid

The definition of the view and materialized table, status table all of the steps defined above are same as they are part of the VDB. Follow same steps above and deploy the VDB in the embedded Teiid as shown below.

```
EmbeddedServer server = new EmbeddedServer();
…
server.addConnectionFactory("name", Object);
…
server.addTranslator("name", ExecutionFactory);
EmbeddedConfiguration config = new EmbeddedConfiguration();
```

```
config.setTransactionManager(EmbeddedHelper.getTransactionManager());
server.start(config);
server.deployVDB("matView-vdb.xml");
```

An Usage Example

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,
  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,
  PRIMARY KEY (VDBName, VDBVersion, SchemaName, Name)
) OPTIONS (UPDATABLE true);
```

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.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<vdb name="example" version="1">
    <model name="Source">
        <property name="importer.useFullSchemaName" value="false" />
        <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,
            PRIMARY KEY (VDBName, VDBVersion, SchemaName, Name)
          );
        ]]>
        </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
- Loading And Refreshing
  - Using System Procedure
  - Using TTL Snapshot Refresh
- Updatable
- Indexes
- Clustering Considerations

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		true	VDB

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

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 on the other hand default to the vdb scope. For materialized views explicitly setting the session or user scopes will result in a non-replicated session scoped materialized view.

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.

Client Developer’s Guide

This guide intended for developers that are trying to write 3rd party applications that interact with Teiid. This will guide you through connection mechanisms, extensions to JDBC API, ODBC, SSL etc. 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 here http://teiid.jboss.org/basics/

JDBC Support

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

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

Generated Keys

Teiid supports returning 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	Generated Keys is not supported 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 any Virtual Database (VDB) deployed on a Teiid Server. The Teiid JDBC API is compatible with the JDBC 4.0 specification; however, it does not fully support all methods. Advanced features, such as updatable result sets or SQL3 data types, are not supported.

Java client applications connecting to a Teiid Server will need to use Java 1.6 JDK. Previous versions of Java are not supported.

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

1. Install the Teiid Server. See the "Admin Guide" for instructions.
2. Build a Virtual Database (VDB). You can build a VDB without a Designer, or you can use the Eclipse based GUI tool Designer. 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.

Now that you have the VDB deployed in the Teiid Server, client applications can connect to the Teiid Server and issue SQL queries against deployed VDB using JDBC API. If you are new to JDBC, see Java's documentation about JDBC. Teiid ships with teiid-9.2.0.Final-jdbc.jar that can be found in the downloads.

You can also obtain the Teiid JDBC from the JBoss Public Maven Repository http://repository.jboss.org/nexus/content/groups/public/ using the coordinates:

```
<dependency>
  <groupId>org.jboss.teiid</groupId>
  <artifactId>teiid</artifactId>
  <classifier>jdbc</classifier>
  <version>9.2.0.Final</version>
</dependency>
```

Against

Main 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.

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 supported 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 local in-VM connection, omit the protocol and host/port.

URL Connection Properties

The following table shows all the supported connection properties that can used 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 identify the connections
`FetchSize`	`int`	Size of the resultset; The default size if 500. ⇐0 indicates that the default should be used.
`partialResultsMode`	`boolean`	Enable/disable support partial results mode. Default false. See the Partial Results Mode section.

`autoCommitTxn`	`String`	Only applies only when "autoCommit" is set to "true". This determines how a executed command needs to be transactionally wrapped inside the Teiid 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 more than one source it automatically uses distributed transaction. Transactions for more information.
`disableLocalTxn`	`boolean`	If "true", the autoCommit setting, commit and rollback will be ignored for 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 default, true, parses dobuled quoted entries as identifiers. If set to false, then double quoted 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 communication exception. Default false. This is typically not needed when connections are managed, as the connection can be purged from the pool.
`SHOWPLAN`	`String`	(typically not set as a connection property) Can be ON, OFF,DEBUG; • ON returns the query plan along with the results • DEBUG additionally prints the query planner debug information in the log and returns it with the results. Both the plan and the log are available through JDBC API extensions. • Default OFF.
`NoExec`	`String`	(typically not set as a connection property) Can be ON, OFF; ON prevents query execution, but parsing and planning will still occur. Default OFF.
`PassthroughAuthentication`	`boolean`	Only applies to "local" connections. When this option is set to "true", then Teiid looks for already authenticated security context on the calling thread. If one found it uses that users credentials to create session. Teiid also verifies that the same user is using this connection during the life of the connection. if it finds a different security context on the calling thread,

		it switches the identity on the connection, if the new user is also eligible to log in to Teiid otherwise connection fails to execute.
`useCallingThread`	`boolean`	Only applies to "local" connections. When this option is set to "true" (the default), then the calling thread will be used to process the query. If false, then an engine thread will be used.
`QueryTimeout`	`integer`	Default query timeout in seconds. Must be >= 0. 0 indicates no timeout. Can be overriden by `Statement.setQueryTimeout` . Default 0.
`useJDBC4ColumnNameAndLabelSemantics`	`boolean`	A change was made in JDBC4 to return unaliased column names as the ResultSetMetadata column name. Prior to this, if a column alias were used it was returned as the column name. Setting this property to false will enable backwards compatibility when JDBC3 and older support is still required. Defaults to true.
`jaasName`	`String`	JAAS configuration name. Only applies when configuring a GSS authentication. Defaults to Teiid. See the Security Guide for configuration required for GSS.
`kerberosServicePrincipleName`	`String`	Kerberos authenticated principle name. Only applies when configuring a GSS authentication. See the Security Guide for configuration required for GSS
`encryptRequest`	`boolean`	Only applies to non-SSL socket connections. When "true" the request message and any associate payload will be encrypted using the connection cryptor. Default false.
`disableResultSetFetchSize`	`boolean`	In some situations tooling may choose undesirable fetch sizes for processing results. Set to true to disable honoring ResultSet.setFetchSize. Default false.
`loginTimeout`	`integer`	The login timeout in seconds. Must be >= 0. 0 indicates no specific timeout, but other timeouts may apply. If a connection cannot be created in approximately the the timeout value an exception will be thrown. A default of 0 does not mean that the login will wait indefinitely. Typically is an active vdb cannot be found the login will fail at that time. Local connections that specify a vdb version however can wait by default for up to org.teiid.clientVdbLoadTimeoutMillis.
`reportAsViews`	`boolean`	If DatabaseMetaData will report Teiid views as a VIEW table type. If false 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 supports the XA protocol, XA transactions will be extended to Teiid sources that also 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
`LoadBalance`	`boolean`	Set to false to disable the default load balancing behavior of selecting a new server when a pooled connection is returned to the pool.

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.

1. 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. 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.

Load Balancing

Post connection load balancing can be utilized in one of two ways. First if you are using `TeiidDataSource` and the Connections returned by Teiid `PooledConnections` have their `close` method called, then a new server instance will be selected automatically. However when using driver based connections or even when using `TeiidDataSource` in a connection pool (such as WildFly), the automatic load balancing will not happen. Second you can explicitly trigger load balancing through the use of the set statement:

```
SET NEWINSTANCE TRUE
```

Typically you will not need want to issue this statement manually, but instead use it as the connection test query on your DataSource configuration.

WildFly DataSource With Post Connection Load Balancing

```
<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>
                <check-valid-connection-sql>SET NEWINSTANCE TRUE</check-valid-connection-sql>
            </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>
```

Teiid by default maintians a pool of extra socket connections that are reused. For load balancing, this reduces the potential cost of switching a connection to another server instance. The default setting is to maintain 16 connections (can be set via `org.teiid.sockets.maxCachedInstances` in a teiid-client-settings.properties file). If you're client is connecting to large numbers of Teiid instances and you're using post connection time load balancing, then consider increasing the number of cached instances. You may either set an analogous system property or create a teiid-client-settings.properties (see the teiid-client-settings.orig.properties file in the client jar) file and place it into the classpath ahead of the client jar.

Note	Session level temporary tables, currently running transactions, session level cache entries, and PreparedPlans for a given session will not be available on other cluster members. Therefore, it is recommended that post connection time load balancing is only used when the logical connection could have been closed, but the actual connection is reused (the typical connection pool pattern).

Advanced Configuration

Server discovery, load balancing, fail over, retry, retry delay, etc. may be customize if the default policy is not sufficient. See the `org.teiid.net.socket.ServerDiscovery` interface and default implementaion `org.teiid.net.socket.UrlServerDiscovery` for how to start with your customization. The `UrlServerDiscovery` implemenation provides the following: discovery of servers from the URL hosts (DataSource server/alternativeServers), random selection for load balancing and failover, 1 connection attempt per host, no biasing, black listing, or other advanced features. Typically you'll want to extend the `UrlServerDiscovery` so that it can be used as the fallback strategy and to only implement the necessary changed methods. It's important to consider that 1 `ServerDiscovery` instance will be created for each connection. Any sharing of information between instances should be done through static state or some other shared lookup.

Your customized server discovery class will then need to be referenced by the discoveryStategy connection/DataSource property by its full class name.

You may also 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.

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 supported 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 Teiid low level and SSL socket 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. For reference the properties are:

```
########################################
# 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

#
# The max number of cached server instances
# to reuse. A server instance is a connected
# socket to a particular cluster member with
# client encryption and or SSL already established.
#
# Caching instances helps in 2 circumstances:
#  - when Connection pooling is not being used.
#  - load-balancing performance to a cluster
#    when using Connection pooling of the DataSource.
#
# This value should typically be a multiple of the
# cluster size.
#
# Set to 0 to disable instance caching.
#

org.teiid.sockets.maxCachedInstances=16

#
# 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


#
# Set to true to disable client ping.
# Default value is 'false' that means client ping is enable. The client ping keeps the relevant server from ter
minating sessions.
# If the default load balancing strategy is used, then client ping is necessary.
#
org.teiid.sockets.DisablePing=false
```

Note	All properties listed in "teiid-client-settings.properties" can also be set as System 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 (see the Admin Guide), and are purged by the least recently used (LRU).
- 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)).

ResultSet Limitations

- TYPE_SCROLL_SENSITIVE is not supported.
- `UPDATABLE` ResultSets are not supported.
- Returning multiple ResultSets from Procedure execution is not supported.

JDBC Extensions

These are custom extensions to JDBC API from Teiid to support 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

The Teiid Server supports a "partial results" query mode. This mode changes the behavior of the query processor so 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 resource adaptor.

Note	Since Teiid supports cursoring before the 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. See the Developer's Guide, in particular the section on ReusableExecutions for more.

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 unwap 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) support 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.

Unsupported JDBC Methods

Based upon the JDBC in JDK 1.6, this appendix details only those JDBC methods that Teiid does not support. Unless specified below, Teiid supports 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 supported. If an argument is listed then only those forms of the method specified are not supported.

Unsupported Classes and Methods in "java.sql"

Class name	Methods
`Blob`	[source,java] ---- getBinaryStream(long, long) - throws SQLFeatureNotSupportedException setBinaryStream(long) - - throws SQLFeatureNotSupportedException setBytes - - throws SQLFeatureNotSupportedException truncate(long) - throws SQLFeatureNotSupportedException ----
`CallableStatement`	[source,java] ---- 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`	[source,java] ---- getCharacterStream(long arg0, long arg1) - throws SQLFeatureNotSupportedException setAsciiStream(long arg0) - throws SQLFeatureNotSupportedException setCharacterStream(long arg0) - throws SQLFeatureNotSupportedException setString - throws SQLFeatureNotSupportedException truncate - throws SQLFeatureNotSupportedException ----
`Connection`	[source,java] ---- 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`	[source,java] ---- getAttributes - throws SQLFeatureNotSupportedException getClientInfoProperties - throws SQLFeatureNotSupportedException getRowIdLifetime - throws SQLFeatureNotSupportedException ----
`NClob`	`Not Supported`
`PreparedStatement`	[source,java] ---- setRef - throws SQLFeatureNotSupportedException setRowId - throws SQLFeatureNotSupportedException setUnicodeStream -

	throws SQLFeatureNotSupportedException ----
`Ref`	`Not Implemented`
`ResultSet`	[source,java] ---- deleteRow - throws SQLFeatureNotSupportedException getHoldability - throws SQLFeatureNotSupportedException getObject(**, Map<String, Class<?>> map) - throws SQLFeatureNotSupportedException 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`

Unsupported Classes and Methods in "javax.sql"

Class name	Methods
`RowSet*`	`Not Supported`

ODBC Support

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. While many PostgreSQL constructs are supported the default behavior for queries matches Teiid's expectations. See System Properties for optional properties that further emulate pgsql handling.

If you do not wish to globally enable org.teiid.backslashDefaultMatchEscape, then have your ODBC client issue "select cast(teiid_session_set('backslashDefaultMatchEscape', true) as boolean)" to alter the default behavior for just the current session.

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 Teiid supports plain text password authentication for ODBC. If the client/server are not configured to use SSL or GSS authentication, the password will be sent in plain text over the network. If you need secure passwords in transit and are not using SSL, then consider installing a security domain that will accept safe password values from the client (for example encrypted or hashed).

See the Security Guide for details on configuring SSL for and using Kerberos with the pg transport.

For a windows client, see the 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 http://psqlodbc.projects.pgfoundry.org/docs/config.html. When using these properties on the connection string, their property names are defined here http://psqlodbc.projects.pgfoundry.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	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 the VDB itself may take default connection properties for ODBC. Use VDB properties of the form connection.XXX to control things like partial results mode, result set caching, etc.

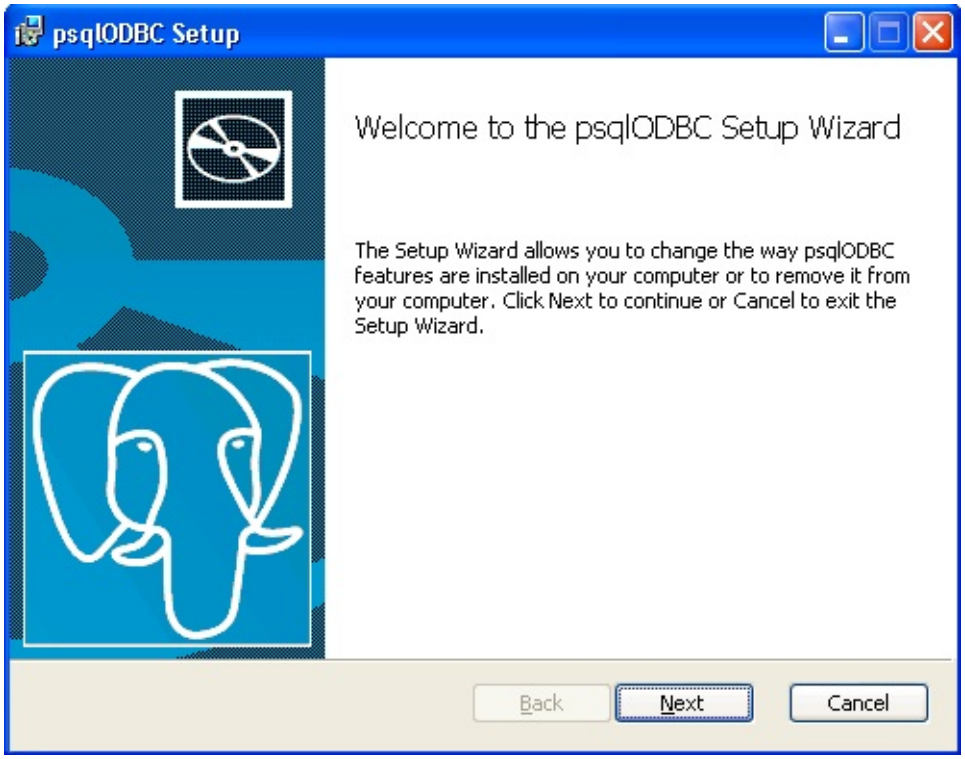
The application name may be set by some clients. If not, you may use a SET statement - "SET application_name name" - to set the name even after the connection is made.

Installing the ODBC Driver Client

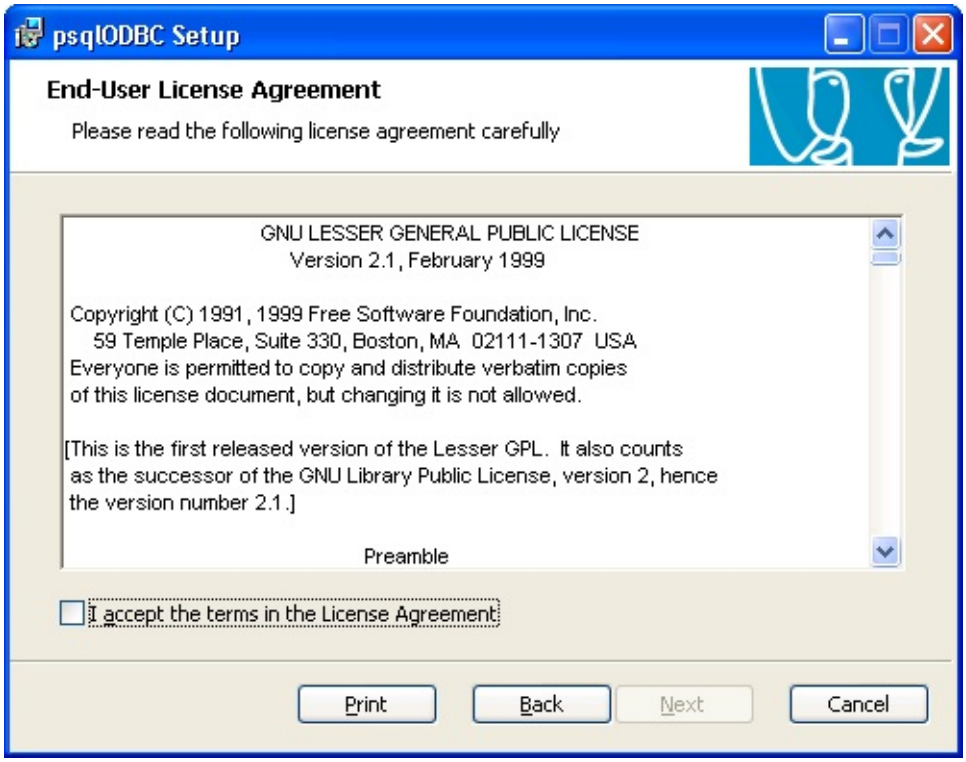
A PostgreSQL ODBC driver needed to make the ODBC connection to Teiid is *not* bundled with the Teiid distribution. The appropriate driver needs be downloaded directly from the PostgreSQL web site. We have tested with *8.04.200* version of the ODBC driver.

Microsoft Windows

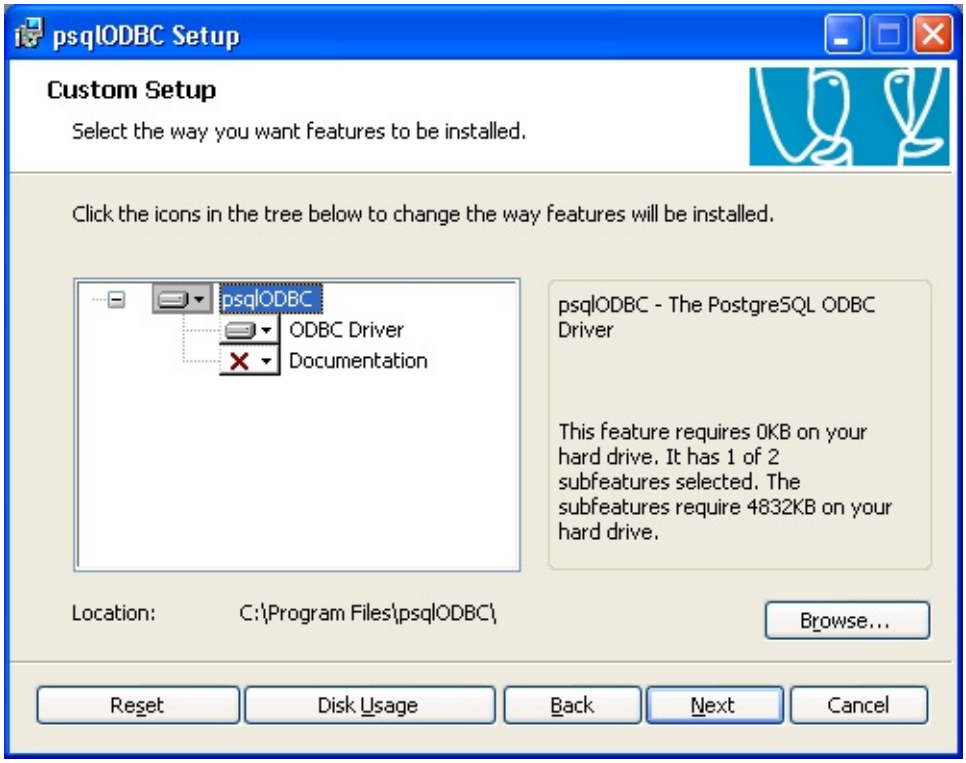
1. Download the ODBC 8.4 driver from PostgreSQL download site. If you are looking for 64-bit Windows driver download the driver from here. Later versions of the driver may be used, but are not considered fully supported.
2. Extract the contents of the ZIP file into a temporary location on your system. 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



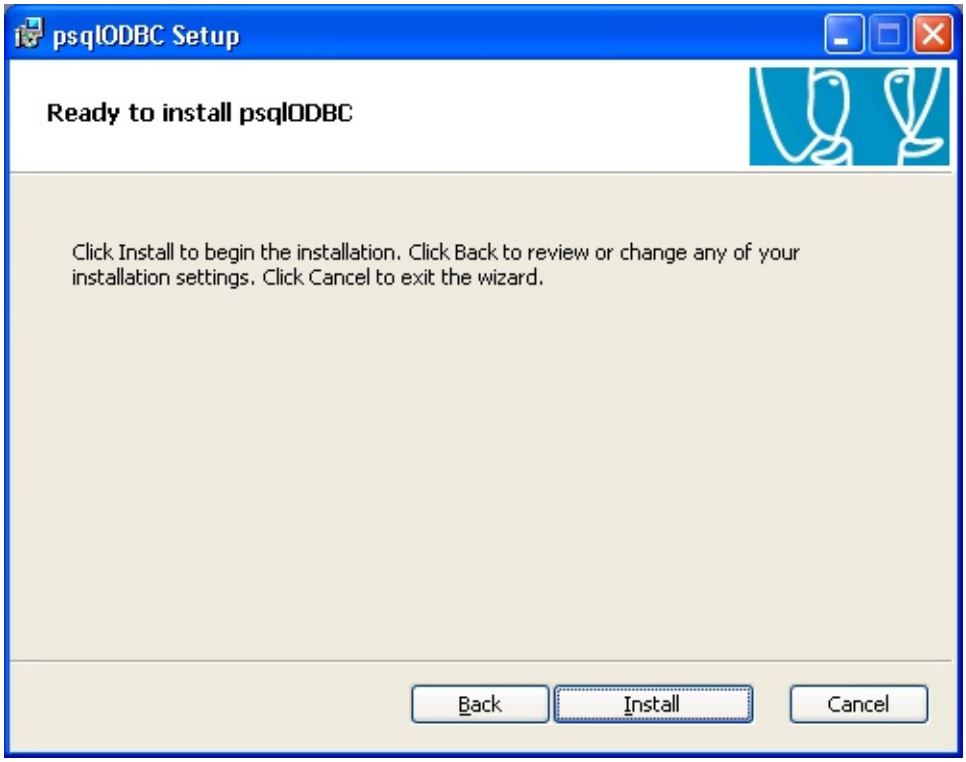
Click "Next". 5. The next step of the wizard displays.



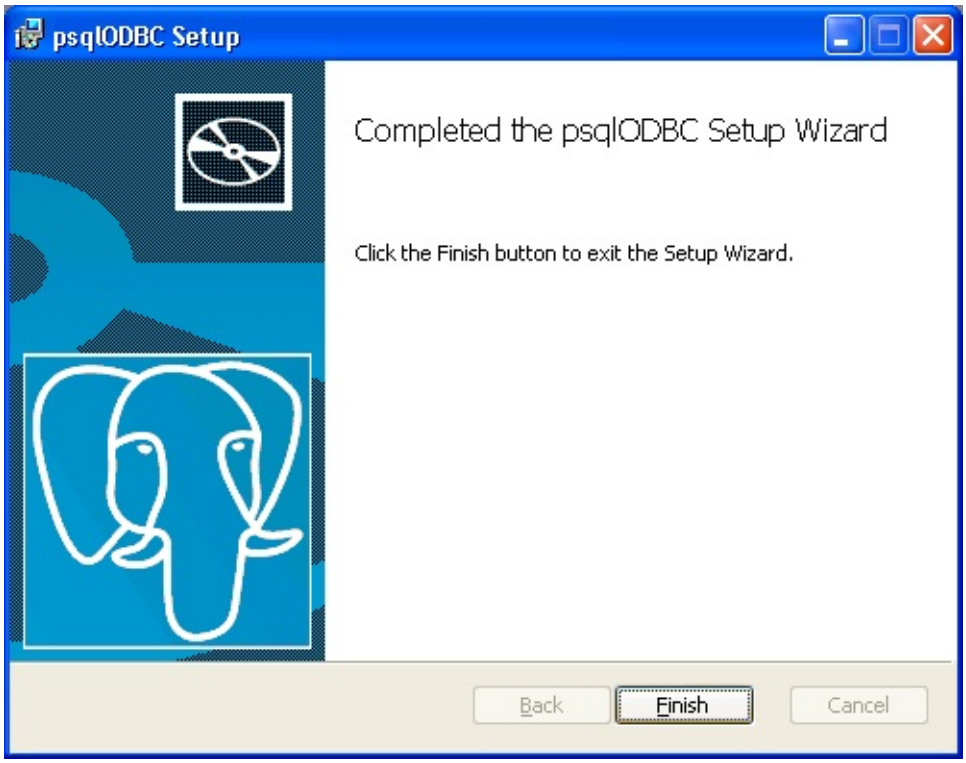
Carefully read it, and check the "I accept the terms in the License Agreement", if you are agreeing to the licensing terms. Then click "Next". 6. The next step of the wizard displays.



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 supported options for a description of the client configuration.

Windows Installation

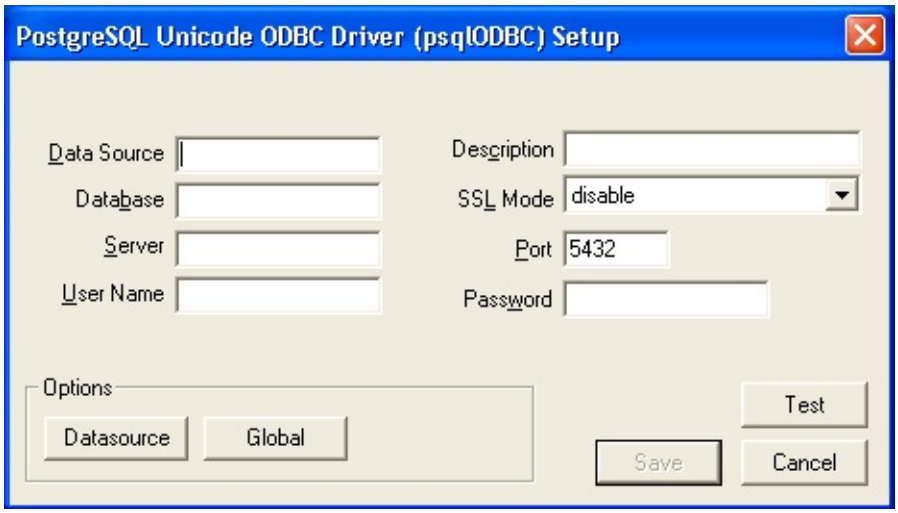
Once you have installed the ODBC Driver Client software on your workstation, you have to configure it to connect to a Teiid Runtime. Note that the following instructions are specific to the Microsoft Windows Platform.

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.

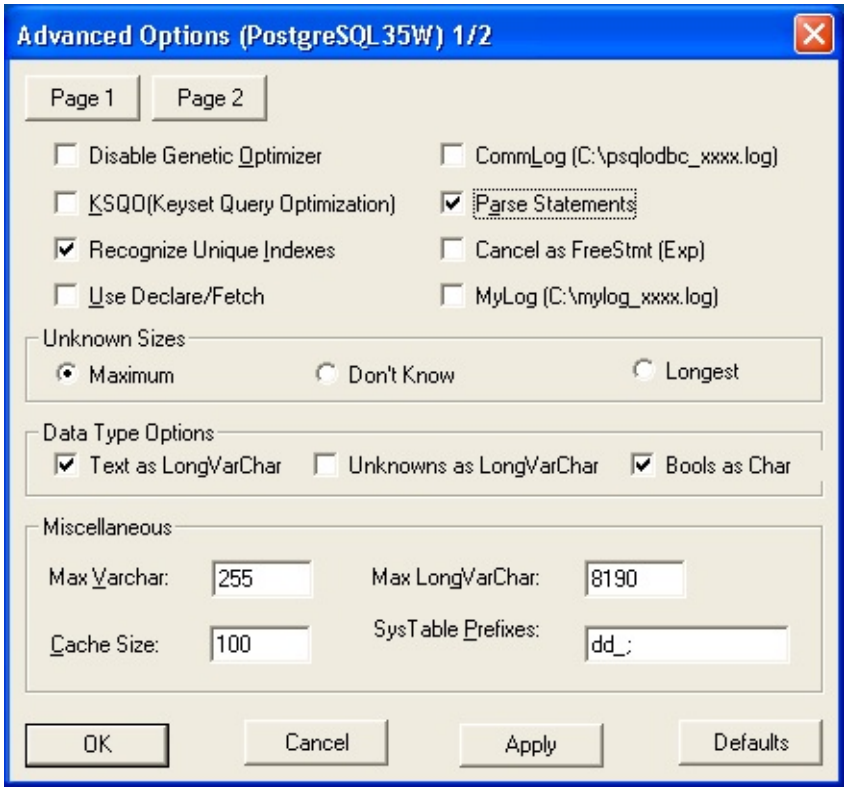
Follow the below steps in creating a data source name (DSN)

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.

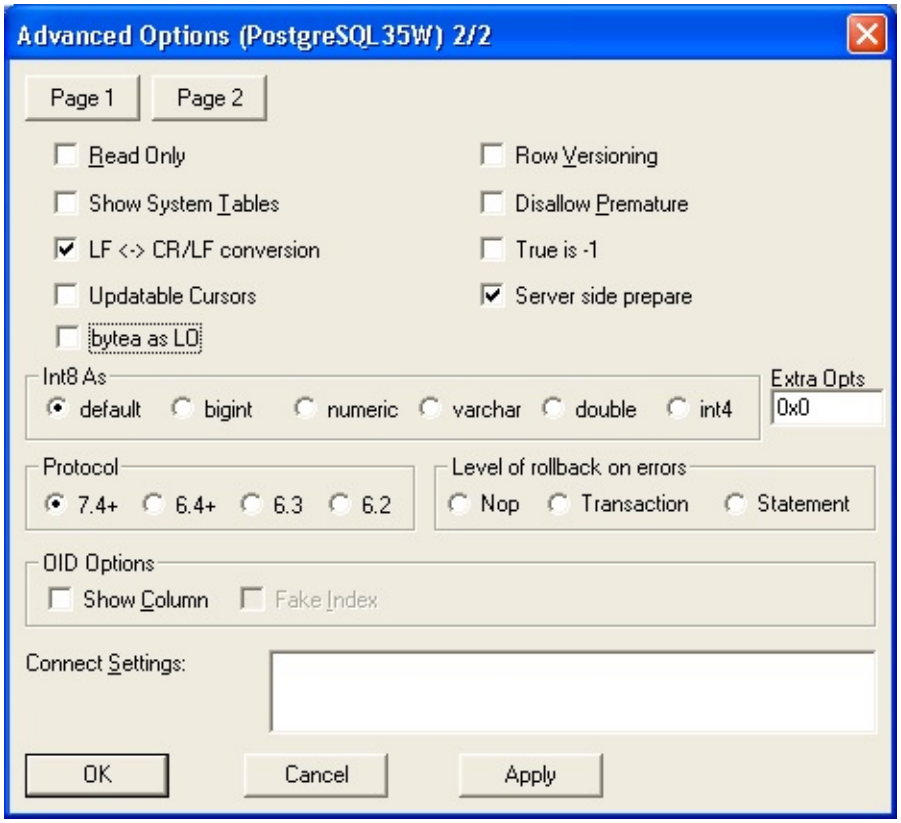


In the *Data Source* Name edit box, type the name you want to assign to this data source. In the *Database* edit box, type the name of the virtual database you want to access through this data source. In the *Server* edit box, type the host name or IP address of your Teiid runtime. If connecting via 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 listenes 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.

1. 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



1. 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 o 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 Teiid supported options.

Configuring Connection Properties with ODBC

When working with ODBC connection, the user can set the connection properties Driver Connection#URL Connection Properties that are available in Teiid by executing the command like below.

```
SET <property-name> TO <property-value>
```

for example to turn on the result set caching you can issue

```
SET resultSetCacheMode TO 'true'
```

Another option is to set this as VDB property in vdb.xml file as

```
<vdb name="...">
    <property name="connection.resultSetCacheMode" value="true"/>
    ...
</vdb>
```

OData Support

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 Support for OData

When a user successfully deploys a VDB into a Teiid Server, the OData protocol support 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.

Teiid provides OData Version 4.0 support. 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 Support

Teiid strives to be compliant with the OData specification. The rest of this chapter highlight some specifics of OData and Teiid's support, but you should also consult the specification.

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:

```
http://localhost:8080/odata/northwind/NW/customers
```

this would be akin to making a JDBC/ODBC connection and issuing the 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/odata/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/odata/northwind/NW/customers?$format=JSON
```

Query options can be combined as needed. For example format with a filter:

```
http://localhost:8080/odata/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/odata/northwind/NW/customers(1234)/customer_fk?$filter=orderdate gt datetime'2012-12-31T21:23:38Z'
```

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.

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/odata/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 'xxx'."}}
```

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". Users are expected to have the **odata** role. Be sure to create user with this role when you are using add-user.sh script to create a new user.

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 can also support Kerberos, SAML and OAuth2 authentications, for configuring the these security schemes please 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
	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.	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

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 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 potion, however the results are not guaranteed to be same as the underlying sources may have been updated with new information meanwhile.

Limitations

The following feature limitations currently apply.

- search is not supported
- delta processing is not supported
- data-aggregation extension to specification is not supported.
- $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). Every VDB, that is deployed in an ACTIVE state in Teiid server exposes its metadata in CSDL format. For example if you want retrieve metadata for your vdb *northwind*, you need to issue a query like

```
http://localhost:8080/odata/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, Action Import
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 dropeed out of the $metadata

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

- Identifier generation based upon table values, such as the hilo generator, require that the identifier table(s) be exposed through Teiid.
- GUID and Identity (using generated key retrieval) identifier generation strategy are directly supported.

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 support 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 show how Hibernate on top of Teiid. Both Hibernate and Eclipselink are fully support JSR-317 (JPA 2.0), 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.
- Your Teiid installation should already be setup for ODBC access. This allows the built-in support of 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 the PostGIS function ST_Estimated_Extent is not supported by Teiid and the execution will be shown in the logs as an error when selecting to compute the bounding box from the data.

Additional Considerations

- If you are integrating a PostgreSQL source, you must not re-expose the geometry_columns table. This is because GeoServer makes unqualified queries that reference geometry_columns and the query should resolve against the Teiid system table instead.

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 support 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 may contain messages related to information_schema.tables - this is to determine if the qgis_editor_widget_styles table exists. That is currently not supported.

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
`PROP_XML_FORMAT / XMLFormat`	Determines the formatting of XML documents returned by XML document models. See the XML extensions#Document formatting section.
`PROP_XML_VALIDATION / XMLValidation`	Determines whether XML documents returned by XML document models will be validated against their schema after processing. See the Reference Guide's "XML SELECT Command" chapter and "document validation" section.
`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.

XML extensions

The XML extensions apply on to XML resutls from queries to XML document models, and not to XML produced by SQL/XML or read from some other source.

Document formatting

The PROP_XML_FORMAT execution property can be set to modify the way that XML documents are formatted from XML document models. Valid values for the constant are defined in the same ExecutionProperties interface:

1. `XML_TREE_FORMAT` - Returns a version of the XML formatted for display. The XML will use line breaks and tabs as appropriate to format the XML as a tree. This format is slower due to the formatting time and the larger document size.
2. `XML_COMPACT_FORMAT` - Returns a version of the XML formatted for optimal performance. The XML is a single long string without any unnecessary white space.
3. Not Set - If no format is set, the formatting flag on the XML document in the original model is honored. This may produce either the "tree" or "compact" form of the document depending on the document setting.

Schema validation

The `PROP_XML_VALIDATION` execution property can be set to indicate that the server should validate XML document model documents against their schema before returning them to the client. If schema validation is on, then the server send a SQLWarning if the document does not conform to the schema it is associated with. Using schema validation will reduce the performance of your XML queries.

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.

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 supports three types of transactions from a client perspective:

1. Global
2. Local
3. Request Level

All are implemented by the Teiid Server 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. Care should be taken to ensure that targeted sources 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) Support

The underlying resource adaptors that represent 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 participate in the distributed transaction. However, the source is still eligible to participate in data integration with out the XA support.

The participation in the XA transaction is automatically determined based on the resource adaptors 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, 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.

Connector Environment Setup

To setup the environment for developing a custom connector, you have 2 options:

1. Manually setup the build environment - structure, framework classes, and resources.
2. Use the Teiid Connector Archetype template to generate the initial project.

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:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <modelVersion>4.0.0</modelVersion>
   <artifactId>connector-{name}</artifactId>
   <groupId>org.company.project</groupId>
   <name>Name Connector</name>
   <packaging>rar</packaging>
   <description>This connector is a sample</description>

   <dependencies>
      <dependency>
         <groupId>org.jboss.teiid</groupId>
         <artifactId>teiid-api</artifactId>
         <scope>provided</scope>
      </dependency>
      <dependency>
         <groupId>org.jboss.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>

</project>
```

Where the ${teiid-version} property should be set to the expected version, such as 9.2.0.Final. You can find Teiid artifacts in the JBoss maven repository. The $\{version.connector.api} version last used was 1.5.

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: https://repository.jboss.org/nexus/content/repositories/releases/ then click OK to return
- Enter "teiid" in the filter to see the Teiid arche types.
- Select the connector-archetype v8.7.1, 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

To create a custom connector project from the command line, you can use the following template command:

```
mvn archetype:generate   -DarchethypeRepository=https://repository.jboss.org/nexus/content/repositories/releases/   \
  -DarchetypeGroupId=org.jboss.teiid.arche-types               \
  -DarchetypeArtifactId=connector-archetype          \
  -DarchetypeVersion=8.7.1               \
  -DgroupId=${groupId}                  \
  -DartifactId=connector-${connector-name}  \
  -Dpackage=org.teiid.resource.adapter.${connector-name}    \
  -Dversion=${teiid.version}    \
  -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 of 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-name  -  (user defined) the name (type) of the new connector project, used to create the java cla
ss names and rar
   -Dvendor-name     -  name of the Vendor for the data source, updates the rar
   -Dteiid-version   -  the Teiid version the connector will depend upon
```

The following is an example:

```
mvn archetype:generate   -DarchethypeRepository=https://repository.jboss.org/nexus/content/repositories/release
s/                                \
   -DarchetypeGroupId=org.jboss.teiid.arche-types              \
   -DarchetypeArtifactId=connector-archetype          \
   -DarchetypeVersion=8.7.1      \
   -DgroupId=org.jboss.teiid.connectors      \
   -Dpackage=org.teiid.resource.adapter.myType    \
   -DartifactId=connector-myType \
   -Dversion=0.0.1-SNAPSHOT    \
   -Dconnector-name=myType   \
   -Dvendor-name=MyVendor    \
   -Dteiid-version=8.7.0.Final
```

When executed, you will be asked to confirm the properties

```
Confirm properties configuration:
groupId: org.jboss.teiid.connectors
artifactId: connector-myType
version: 0.0.1-SNAPSHOT
package: org.teiid.resource.adapter.myType
connector-name: myType
vendor-name: MyVendor
teiid-version: 8.7.0.Final
 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:

```
<dependencies>
    <dependency>
        <groupId>org.jboss.teiid</groupId>
        <artifactId>teiid-api</artifactId>
        <version>${teiid-version}</version>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>org.jboss.teiid</groupId>
        <artifactId>teiid-common-core</artifactId>
        <version>${teiid-version}</version>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>javax.resource</groupId>
        <artifactId>connector-api</artifactId>
        <version>${version.connector.api}</version>
        <scope>provided</scope>
    </dependency>
</dependencies>
```

Where the ${teiid-version} property should be set to the expected version, such as 9.2.0.Final. You can find Teiid artifacts in the JBoss maven repository. The $\{version.connector.api} version last used was 1.5.

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: https://repository.jboss.org/nexus/content/repositories/releases/ then click OK to return
- Enter "teiid" in the filter to see the Teiid arche types.
- Select the translator-archetype v8.7.1, 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

To create a custom translator project from the command line, you can use the following template command:

```
mvn archetype:generate                                \
  -DarchethypeRepository=https://repository.jboss.org/nexus/content/repositories/releases/   \
  -DarchetypeGroupId=org.jboss.teiid.arche-types              \
  -DarchetypeArtifactId=translator-archetype         \
  -DarchetypeVersion=8.7.1               \
  -DgroupId=${groupId}                  \
  -DartifactId=translator-${translator-name}    \
  -Dpackage=org.teiid.translator.${translator-name}    \
  -Dversion=${version}    \
  -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-name -  (user defined) the name (type) of the new translator project, used to create the java cl
ass names
   -Dteiid-version   -  the Teiid version the connector will depend upon
```

The following is an example to execute:

```
mvn archetype:generate                                \
  -DarchethypeRepository=https://repository.jboss.org/nexus/content/repositories/releases/   \
  -DarchetypeGroupId=org.jboss.teiid.arche-types               \
  -DarchetypeArtifactId=translator-archetype         \
  -DarchetypeVersion=8.7.1      \
  -DgroupId=org.jboss.teiid.connector   \
  -DartifactId=translator-myType    \
  -Dpackage=org.teiid.translator.myType    \
  -Dversion=0.0.1-SNAPSHOT  \
  -Dtranslator-name=MyType  \
  -Dteiid-version=8.7.0.Final
```

When executed, you will be asked to confirm the properties

```
Confirm properties configuration:
groupId: org.jboss.teiid.connector
artifactId: translator-myType
version: 0.0.1-SNAPSHOT
package: org.teiid.translator.myType
teiid-version: 8.7.0.Final
translator-name: MyType
 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 BatchedCommand, 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 elements) 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. While builing your VDB in the Designer, you can define what called "Extension Model", that defines any number of arbitary properties on a model and its objects. 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 elements 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 elements 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}}s 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 libraries. This Execution Factory needs to define/override the following elements.

```
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. This method is used by Designer tooling when the Teiid Connection importer is used. 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 set through Designer when using the "Teiid Connection" importer or the properties 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) 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.
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.
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		Translator 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.
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.
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
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.
ElementaryOlapOperations		Translator supports window functions and analytic functions RANK, DENSE_RANK, and ROW_NUMBER.
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.
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.

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.

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 "vdb.xml" file 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 vdb.xml as

```
<?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.xml file

```
<?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>
```

Note	**"Designer Integration"** - When properties are defined using the annotation mechanism and when using the "Teiid Connection" importer in the Designer, these properties will automatically show up in appropriate wizard for input.

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
```

Note	**"Designer Integration"** - When extended properties are defined using the annotation mechanism, when using "Teiid Connection" importer in the Designer, you do not need to define the "Metadata Extension Defn" in designer and register to to use with your model, the required definitions are automatically downloaded and configured to use. (This feature is not available in current Designer version)

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
- SQL Translation Extension
- Results Translation Extension
- Adding Function Support
- Using FunctionModifiers
- Installing Extensions

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 differnt 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 AdminShell 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 funcitons 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.

When Using Designer

If you are designing your VDB using the Designer, you can define a function on any "source" model, and that function is automatically added as pushdown function when the VDB is deployed. There is no additional need for adding Java code.

Without Designer

If you are not using Designer, see defining the metadata using DDL, you can define your source function in the VDB as

```
<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

```
<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,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.

Metadata in Designer

A user defined function created on any VDB on view model by creating a Function just as a base table. You would require all the information defined on User Defined Functions to create a UDF. Make sure you provide the JAVA code implementation details in the properties dialog for the UDF.

Metadata without Designer

When defining the metadata without Designer, you can define a UDF or UDAF (User Defined Aggregate Function) as shown below.

```
<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: https://repository.jboss.org/nexus/content/repositories/releases/ then click OK to return
- Enter "teiid" in the filter to see the Teiid arche types.
- Select the udf-archetype v9.0.0, 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

To create a custom translator project from the command line, you can use the following template command:

```
mvn archetype:generate       -DarchethypeRepository=https://repository.jboss.org/nexus/content/repositories/releases/                              \
  -DarchetypeGroupId=org.jboss.teiid.arche-types              \
  -DarchetypeArtifactId=udf-archetype          \
  -DarchetypeVersion=9.0.0     \
  -DgroupId=${groupId}                                  \
  -DartifactId=${udf-artifact-id}       \
  -Dpackage=${packageName}    \
  -Dversion=0.0.1-SNAPSHOT      \
  -Dudf-name=${functionName}  \
  -Dmethod-name=${methodName}      \
  -Dmethod-args=${methodArguments}   \
  -Dreturn-type=${returnType}
```

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 created
  -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
 Name
  -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
```

The following is an example to execute:

```
mvn archetype:generate         -DarchethypeRepository=https://repository.jboss.org/nexus/content/repositories/r
eleases/   \                              \
  -DarchetypeGroupId=org.jboss.teiid.arche-types              \
  -DarchetypeArtifactId=udf-archetype          \
  -DarchetypeVersion=9.0.0     \
  -DgroupId=org.teiid.udf   \
  -DartifactId=udf-myFunction    \
  -Dpackage=org.teiid.udf    \
  -Dversion=0.0.1-SNAPSHOT      \
  -Dudf-name=myFunction \
  -Dmethod-name=myFunction      \
  -Dmethod-args='String arg1'   \
  -Dreturn-type=String
```

When executed, you will be asked to confirm the properties

```
[INFO] Archetype repository not defined. Using the one from [org.jboss.teiid.arche-types:udf-archetype:9.0.0] f
ound in catalog local
[INFO] Using property: groupId = org.teiid.udf
[INFO] Using property: artifactId = udf-myFunction
[INFO] Using property: version = 0.0.1-SNAPSHOT
[INFO] Using property: package = org.teiid.udf
[INFO] Using property: method-args = String arg1
[INFO] Using property: method-name = myFunction
[INFO] Using property: return-type = String
[INFO] Using property: udf-name = myFunction
Confirm properties configuration:
groupId: org.teiid.udf
artifactId: udf-myFunction
version: 0.0.1-SNAPSHOT
package: org.teiid.udf
method-args: String arg1
method-name: myFunction
return-type: String
udf-name: myFunction
 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.

AdminAPI

In most circumstances the admin operations will be performed through the admin console or AdminShell tooling, but it is also possible to invoke admin functionality directly in Java 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-9.2.0.Final.jar*. The CommmdLogMessage includes information about vdb, session, command sql, etc. CommandLogMessages are logged at the INFO (user queries), DEBUG (source queries), 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-9.2.0.Final.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 VDB file. In Designer based VDB, you can edit the vdb.xml file in the META-INF directory or use 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 trigger
Operation);
void setInsteadOfTriggerDefinition(String vdbName, String vdbVersion, Table table, Table.TriggerEvent triggerOp
eration, String triggerDefinition);
boolean isInsteadOfTriggerEnabled(String vdbName, String vdbVersion, Table table, Table.TriggerEvent triggerOpe
ration);
void setInsteadOfTriggerEnabled(String vdbName, String vdbVersion, Table table, Table.TriggerEvent triggerOpera
tion, 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 above 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 Setting up the build environment to start development. For Example:

Sample Java Code

```
import org.teiid.metadata.MetadataRepository;
...

package com.something;

public class CustomMetadataRepository extends 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="javax.resource.api"/>
        <module name="org.jboss.teiid.common-core"/>
        <module name="org.jboss.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. 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
    }
}
```

Then build a JAR archive with above implementation class and create a file named *org.teiid.PreParser* in *META-INF/services* directory with contents:

```
com.something.CustomPreParser
```

The the jar has been built, this 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.jboss.teiid.common-core"/>
        <module name="org.jboss.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.
- Use the cli or modify the configuration to set the preparser-module in the Teiid subsystem configuration to the appropriate module name.
- Restart the server

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.

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: https://repository.jboss.org/nexus/content/repositories/releases/ 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

To create a custom preparser project from the command line, you can use the following template command:

```
mvn archetype:generate       -DarchethypeRepository=https://repository.jboss.org/nexus/content/repositories/rel
eases/                                \
  -DarchetypeGroupId=org.jboss.teiid.arche-types              \
  -DarchetypeArtifactId=preparser-archetype          \
  -DarchetypeVersion=9.0.0     \
  -DgroupId=${groupId}                                  \
  -DartifactId=${udf-artifact-id}       \
  -Dpackage=${packageName}    \
  -Dversion=0.0.1-SNAPSHOT      \
  -Dclass-name=${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 preparser project pom.xml
   -Dpackage             -  (user defined) the package structure where the java, module and resource files will
be created
   -Dversion             -  (user defined) the version that the new preparser 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       -  Optional, defaults to 9.0.0.Final
```

- EXAMPLE
- this is an example of the template that can be run:

```
mvn archetype:generate       -DarchethypeRepository=https://repository.jboss.org/nexus/content/repositories/rel
eases/                                \
  -DarchetypeGroupId=org.jboss.teiid.arche-types               \
  -DarchetypeArtifactId=preparser-archetype          \
  -DarchetypeVersion=9.0.0     \
  -DgroupId=org.teiid.preparser   \
  -DartifactId=preparser-myParser    \
  -Dpackage=org.teiid.preparser    \
  -Dversion=0.0.1-SNAPSHOT      \
  -Dclass-name=myPreParser
```

When executed, you will be asked to confirm the package property

```
[INFO] Using property: groupId = org.teiid.preparser
[INFO] Using property: artifactId = preparser-myParser
[INFO] Using property: version = 0.0.1-SNAPSHOT
[INFO] Using property: package = org.teiid.preparser
[INFO] Using property: class-name = myPreParser
[INFO] Using property: teiid-version = 9.0.0.Final
Confirm properties configuration:
groupId: org.teiid.preparser
artifactId: preparser-myParser
version: 0.0.1-SNAPSHOT
package: org.teiid.preparser
class-name: myPreParser
teiid-version: 9.0.0.Final
 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 7+ 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 Classpath
  - Embedded Using Maven
  - OSGI
  - Maven
  - Vulnerable Libraries
- VDB Deployment
- Access from client applications
- Security
  - Example
- Transactions
- AdminApi
- Logging
- Other Differences Between Teiid Embedded and an AS Deployment

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 4.1.6 or compatible, will be needed for specific features - such as utilizing the JDBC translator support for dependent joins using temp tables.

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 depedency management section:

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.jboss.teiid</groupId>
      <artifactId>teiid-parent</artifactId>
      <version>${version.teiid}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
```

OSGI

All Teiid jars can also be deployed as bundles in a OSGI container like Karaf. If you are working with Karaf, a feature.xml file is available in maven repo for your convenience. Usage pattern is below

```
features:addurl mvn:org.jboss.teiid/teiid/9.1.0.Final/xml/karaf-features
features:install -v teiid
```

Maven

If you are trying run Teidd Embedded with Maven based project and using maven to pull artifacts, the `runtime`, `admin`, `connector`, `translator` dependencies are necessary as below

```
<dependency>
    <groupId>org.jboss.teiid</groupId>
    <artifactId>teiid-runtime</artifactId>
</dependency>

<dependency>
    <groupId>org.jboss.teiid</groupId>
    <artifactId>teiid-admin</artifactId>
</dependency>

<dependency>
    <groupId>org.jboss.teiid.connectors</groupId>
    <artifactId>translator-SOURCE</artifactId>
</dependency>

<dependency>
    <groupId>org.jboss.teiid.connectors</groupId>
    <artifactId>connector-SOURCE</artifactId>
</dependency>
```

Vulnerable Libraries

Prior to Teiid 8.13/8.12.2 when using the remote JDBC transport, Teiid embedded could be susceptible to deserialization exploits if it also included most versions of common-collections, older version of groovy, or spring in the classpath - see also this posting for more details on the affected libraries.

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. Teiid Designer 7 and later VDBs are also supported, but are subject to all of the limitations/differences highlighted in this guide.

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.
- The legacy function model is not supported.

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
- How to develop a SecurityHelper
- Embedded Security with UsersRolesLoginModule
- Embedded Security with LdapExtLoginModule

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.jboss.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);
```

Reference Guide

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

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

Before one can delve into Teiid it is very important to learn few basic constructs of Teiid, like what is VDB? what is Model? etc. For that please read the short introduction here http://teiid.jboss.org/basics/

Data Sources

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

The types of data sources that are currently accessible are:

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

Databases

See JDBC Translators for access to:

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

Web Services

See Web Services Translator for access to:

- SOAP
- REST
- Arbitrary HTTP(S)

OData

See the OData Translator

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

- Actian Vector
- Amazon SimpleDB
- Apache Accumulo
- Apache Cassandra DB
- Apache SOLR
- Apache Spark
- 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

- JDG / Infinispan Library Mode
- JDG / Infinispan HotRod Mode
- Intersystems Cache Object Database
- JPA sources

LDAP

See the LDAP Translator for access to:

- RedHat Directory Server
- Active Directory

Files

See the File Translator for use with:

- Delimited/Fixed width
- XML

Spreadsheets

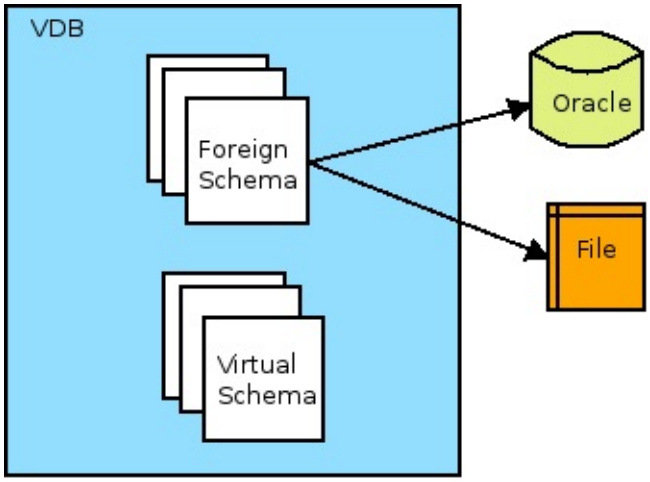
- Excel
- Google Spreadsheet

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

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

Virtual Databases

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



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

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

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

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

Example: 1

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

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

Example: 2

```
CREATE DATABASE my_example VERSION '1.0.0';
USE DATABASE my_example VERSION '1.0.0'
CREATE FOREIGN DATA WRAPPER postgresql;
CREATE SERVER pgsql TYPE 'postgresql-9.4-1201.jdbc41.jar'
    VERSION 'one' FOREIGN DATA WRAPPER postgresql
    OPTIONS (
        "jndi-name" 'java:/postgres-ds'
    );
CREATE SCHEMA test SERVER pgsql;
```

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

Both formats define the same VDB.

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

External Data Sources

A "source schema" component in VDB as shown in above example is a collection schema elements as tables, procedures and functions that represent an external data source's metadata locally. However in the above example, it did not define any such schema elements, however details of connection to the external data source were provided through "jndi-name", which is a named connection reference to a external data source.

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

Resource Adapter

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

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

Translator

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

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

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

Developing a Virtual Database

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

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

- vdb.xml : In this file format, you can use combination of XML elements and DDL elements to define the metadata.
- vdb.ddl : In this file format, you can use strictly DDL using SQL-MED (with few custom extensions) to define the metadata. This can be viewed as next version to the vdb.xml.
- Designer : Designer provides a graphical UI, that user can use to design a VDB. Using this, user can interactively design tables, views, procedures, functions etc. Designer typically generates a .vdb (zip) file artifact, however it can also export this as -vdb.xml file. Support to export as vdb.ddl is coming soon.

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

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

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

Steps to follow in developing a VDB

This will walk through developing a DDL based VDB.

Step 1: Pick Name and Version

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

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

Step 2: Configuring a Source(s)

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

Step 2A: Find the Translator

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

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

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

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

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

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

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

Step 2C: Create a Connection to External Source

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

From previous example this represents

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

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

Step 3: Create Source Schema

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

From previous example this represents

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

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

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

Step 3A: Import Metadata

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

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

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

Step 3B: Define Metadata using DDL

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

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

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

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

Step 5: Create Virtual Views

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

```
CREATE VIRTUAL SCHEMA reports;
```

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

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

Step 6: Deploy the VDB

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

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

Step 7: Client Access

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

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

vdb.xml

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

See also link:xml_deployment_mode.adoc

VDB Zip Deployment

For more complicated scenarios you are not limited to just an xml/ddl file deployment. A full zip file similar to a Designer VDB may also be deployed. 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.
- For backwards compatibility with Designer VDBs, if any .INDEX file exists the default metadata repository will be assumed to be INDEX.
- Files within the VDB zip are accessible by a Custom Metadata Repository using the `MetadataFactory.getVDBResources()` method, which returns a map of all `VDBResources` in the VDB keyed by absolute path relative to the vdb root. The resources are also available at runtime via the SYSADMIN.VDBResources table.
- The built-in DDL-FILE metadata repository type may be used to define DDL-based metadata in other files within the zip archive. This improves the memory footprint of the vdb metadata and the maintainability of the metadata.

Example VDB Zip Structure

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

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

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

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

DDL VDB

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

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

DDL File Deployment

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

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

Example VDB DDL Template

```
CREATE DATABASE my_example VERSION '1.0.0';
USE DATABASE my_example VERSION '1.0.0'
CREATE FOREIGN DATA WRAPPER postgresql;
CREATE SERVER pgsql TYPE 'postgresql-9.4-1201.jdbc41.jar'
    VERSION 'one' FOREIGN DATA WRAPPER postgresql
    OPTIONS (
        "jndi-name" 'java:/postgres-ds'
    );
CREATE SCHEMA test SERVER pgsql;
IMPORT FOREIGN SCHEMA public FROM SERVER pgsql INTO test
    OPTIONS(
        importer.useFullSchemaName false,
        importer.tableTypes 'TABLE,VIEW'
);
```

DDL File Format

Step 1: 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);
```

you can add additional OPTIONS at later time using below syntax.

```
ALTER  DATABASE <name> OPTIONS (ADD|SET|DROP <key-value>)
```

For list database scoped properties see VDB properties

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.

Step 2A: Create a Translator

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. For list of available translators, you should look up in Teiid system. Some of the examples of available translator modules 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 "override" translator. The *OPTIONS* clause is used to provide the "execution-properties" of a specific translator defined in either in {translator-name} or {base-translator-name}. These names **MUST** match with available Translators in the system. link:Translators.adoc[Translators} documents all the available translators.

Example 1: Example creating translator

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

For all available translators see Translators

1. Example 2: Example creating Override Translator

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

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

Additional management support to alter, delete a translator

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

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

Step 2C: Create a Connection To External Source

Before you can create a connection to the data source, you must either have a JDBC driver (Type 4) that can connect to the data source, or Teiid system must have provided a resource adapter (RAR) file to enable connection to the data source. If you are using the JDBC driver file this should have already been deployed to the Teiid system, or made it available on the classpath in the case of the Teiid Embedded. This is Step 2B from earlier, there is currently no DDL mechanism to deploy the external drivers.

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

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

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

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

For all available translators see Translators

Example 3: creating a data source connection to Postgres database

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

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

Name	Description
jndi-name	Jndi name of the datasource

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

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

The below shows an example connection with resource adapter.

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

```
CREATE SERVER marketdata TYPE 'file'
```

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

For all available data sources see data sources

Additional management support to alter/delete a connection.

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

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

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

Step 3 & 5: Create SCHEMA in VDB

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

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

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

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

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

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

Name	Description
VISIBILITY	Is Schema visible during metadata interrogation

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

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

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

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

When working with existing VDB, or when editing multiple Schema's in a working VDB, one needs to setup context of Schema he/she wants to modify, as some of the DDL commands are context sensitive as to which Schema they belong. To be able to establish the context that you are working with certain Schema use following command.

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

then you will be add/drop/alter any other Schema elements.

Step 4: 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} [ <import qualifications> ]
    FROM SERVER {server-name}
    INTO {schema-name}
    OPTIONS (<options-clause>)

<import qualifications> ::=
    LIMIT TO (<table name list>) | EXCEPT (<table name list>)

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

foreign-schema-name : Name of schema in external data source to import. Typically most databases are tied to a schema name, like "public", "dbo" or name of the database. If you are working with non-relational source, you can provide a dummy value here. server-name: name of the server created above to import metadata from. schema-name: Name schema element created above to import metadata into. import qualifications : using this you can limit your import of the Tables from foreign datasource specified to this list. options-clause : The "importer" properties that can be used to refine the import process behavior of the metadata. Each Translator defines a set of "importer" properties with their documentation or through extension properties.

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

Example 6

```
-- import from native database
IMPORT FOREIGN SCHEMA public
    LIMIT TO customers, orders
    FROM SERVER pgsql
    INTO test

-- in archive based vdbs(.vdb) you can provide each schema in a separate file and
pull them in main vdb.ddl file as
IMPORT FOREIGN SCHEMA public
    FROM REPOSITORY DDL-FILE
    INTO test OPTIONS ("ddl-file" '/path/to/schema.ddl')
```

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

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

Importing another Virtual Database (VDB Reuse)

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

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

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

Data Roles

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

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

BNF for Create Data Role

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

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

Example: Create Data Role

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

CREATE ROLE readOnly WITH ANY AUTHENTICATED;
```

Now to Setting Permissions, see Permissions for more details.

BNF for Grant command

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

<permission-types> ::=
    SELECT | INSERT |  UPDATE | DELETE |
    EXECUTE | LANGUAGE | ALTER | DROP |
    ALL PRIVILEGES | TEMPORARY TABLES)

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

<condition> ::=
```

```
    CONDITION [CONSTRAINT] {expression}
```

- permission-types: Types of permissions to be granted
- grant-resource: This is Schema element in the VDB on which this grant applies to. It can anything between a DATABASE and COLUMN
- schema-name: Name of the schema this resource belongs to
- table-name: Name of the Table/View
- procedure-name: Procedure Name
- column-name: Name of the column
- condition: When present, the {expression} is appended to the WHERE clause of the query
- expression: any valid sql expression, this can include columns from referenced resource
- CONSTRAINT: When this is supplied along with CONDITION, the {expression} is also applied during the INSERT/UPDATE queries. By default CONDITION **only** applies SELECT queries. Also CONSTRAINT does **NOT** apply to VIEWs only FOREIGN TABLES.

Caution	LANGUAGE permission can only be applied on DATABASE

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

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

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

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

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

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

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

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

Example : Restricting rows, ROW BASED SECURITY

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

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

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

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

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

```
GRANT READ ON COLUMN test.Order.amount
```

```
    MASK 'CASE WHEN amount > 1000 THEN 'xxxx' END'
    TO base-role;
```

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

```
GRANT READ ON COLUMN test.Order.amount
    MASK 'xxxx'
    CONDITION 'customerid <> user()'
    TO base-role;
```

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 INDEX for Designer VDBs and NATIVE for non-Designer VDB source models. For all other deployments/models a value must be specified. Built-in types include DDL, NATIVE, INDEX, and DDL-FILE. The usage of the raw text varies with the by type. NATIVE and INDEX (only for Designer VDBs) 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 linke:DDL_Metadata.adoc[Schema Object DDL]. Rest of 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

Translator Element

Attributes

- *name*

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

- *type*

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

Property Elements

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

VDB Reuse

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

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

Example reuse VDB XML

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

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

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

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

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

Virtual Database Related Properties

Properties DATABASE Level

- *cache-metadata*

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

- *query-timeout*

Sets the default query timeout in milliseconds for queries executed against this VDB. 0 indicates that the server default query timeout should be used. Defaults to 0. Will have no effect if the server default query timeout is set to a lesser value. Note that clients can still set their own timeouts that will be managed on the client side.

- *lib*

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

- *security-domain*

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

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

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

- *connection.XXX*

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

```
    <property name="connection.partialResultsMode" value="true" />
```

- *authentication-type*

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

- *password-pattern*

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

- *gss-pattern*

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

- *model.visible*

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

- *include-pg-metadata*

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

- *lazy-invalidate*

By default TTL expiration will be invalidating - see Internal Materialization. Setting lazy-invalidate to true will make ttl refreshes non-invalidating.

- *deployment-name*

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

Properties Schema/Model Level

- *visible*

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

- *multisource*

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

- *multisource.columnName*

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

- *multisource.addColumn*

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

- *allowed-languages*

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

DDL Metadata for Schema Objects

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

Table of Contents

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

Data Types

The BNF for Data Types refer to Data Types

Creating a Foreign Table

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

BNF for Create Table

```
CREATE FOREIGN TABLE {table-name} (
    <table-element> (,<table-element>)*
    [<constraint> (,<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] [NOTNULL] [PRIMARY KEY] [UNIQUE] [INDEX] [DEFAULT {expr}]

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

<columns> ::=
```

```
    ( {column-name} [,{column-name}] )

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

For validating BNF for create table refer to CREATE TABLE

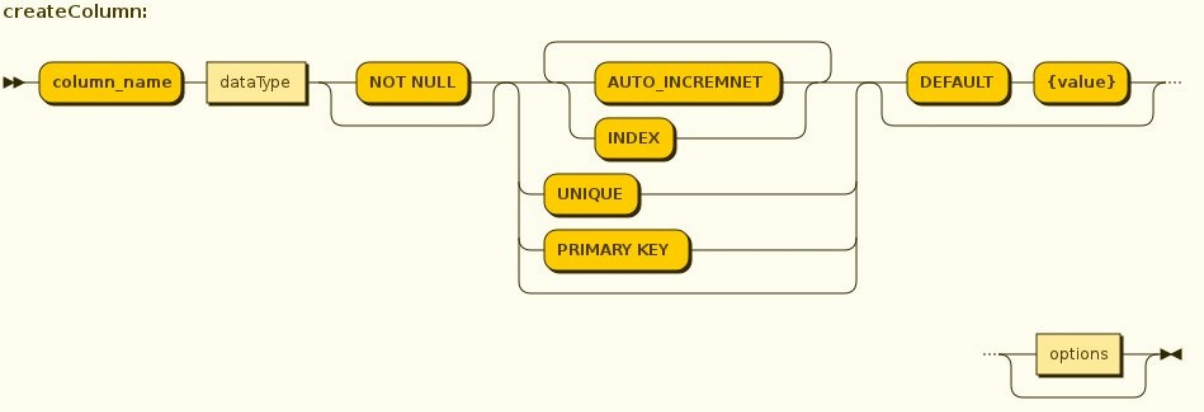
Example 7:Create Foreign Table(Created on PHYSICAL model)

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

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

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

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



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

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

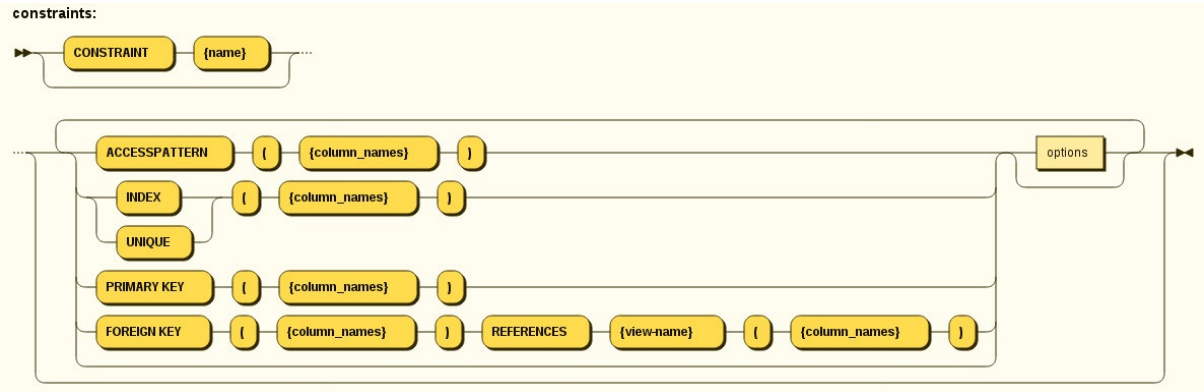
CHAR_OCTET_LENGTH	integer	
ANNOTATION	string	
NATIVE_TYPE	string	
RADIX	integer	
NULL_VALUE_COUNT	long	costing information Number of NULLS in this column
DISTINCT_VALUES	long	costing information Number of distinct values in the column

Columns may also be marked as NOT NULL, auto_increment, and with a DEFAULT value. Currently only string values are supported as the default value. To have the string interpreted as an expression use the extension property teiid_rel:default_handling set to expression.

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

Defining Table CONSTRAINTS

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



CONSTRAINTS are same as one can define on RDBMS.

Example of CONSTRAINTs

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

ALTER TABLE

The BNF for ALTER table, refer to ALTER TABLE

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

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

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

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

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

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

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

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

Create View

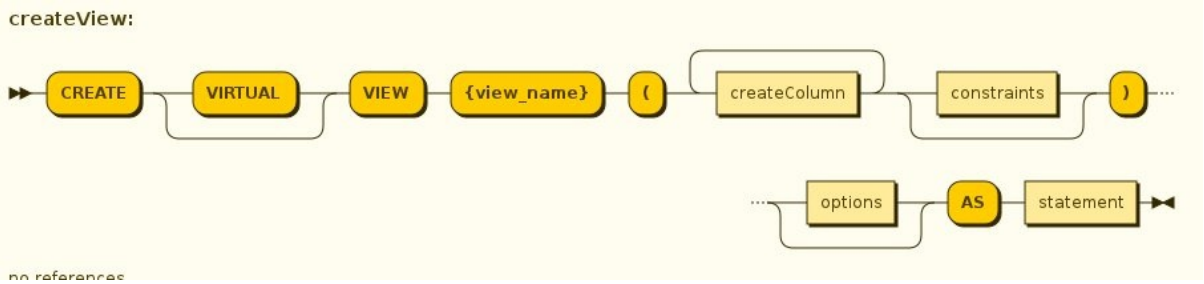
A view is a virtual table. A view contains rows and columns,like a real table. The columns in a view are columns from one or more real tables from the source or other view models. They can also be expressions made up multiple columns, or aggregated columns. When column definitions are not defined on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

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

BNF for Create Table

```
CREATE VIEW {table-name} AS {transformation_query}
    OPTIONS (<options-clause>)

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



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

Property	Data Type or Allowed Values	Description
MATERIALIZED	'TRUE'|'FALSE'	Defines if a table is materialized

MATERIALIZED_TABLE	'table.name'	If this view is being materialized to a external database, this defines the name of the table that is being materialized to

Example:Create View Table(Created on VIRTUAL schema)

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

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

ALTER TABLE

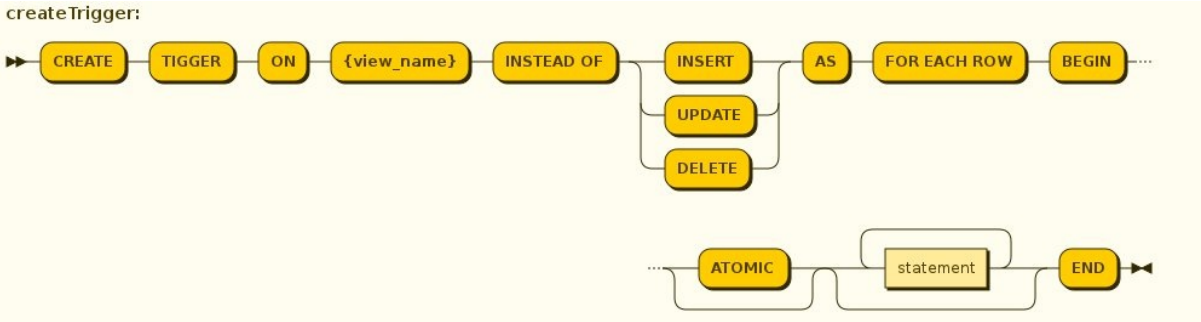
The BNF for ALTER VIEW, refer to ALTER TABLE

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

```
ALTER VIEW CustomerOrders
    AS
    SELECT concat(c.firstname, c.lastname) as name,
        o.saledate as saledate,
        o.amount as amount
  FROM Customer C JOIN Order o ON c.id = o.customerid
  WHERE saledate < TIMESTAMPADD(now(), -1, SQL_TSI_MONTH)
```

INSTEAD OF TRIGGERS On VIEW (Update VIEW)

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



Example:Define instead of trigger on View for INSERT

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

For Update

Example:Define instead of trigger on View for UPDATE

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

While updating you have access to previous and new values of the columns. For more detailed explanation of these update procedures please refer to Update Procedures

AFTER TRIGGERS On Source Tables

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

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

Example:Define AFTER trigger on Customer

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

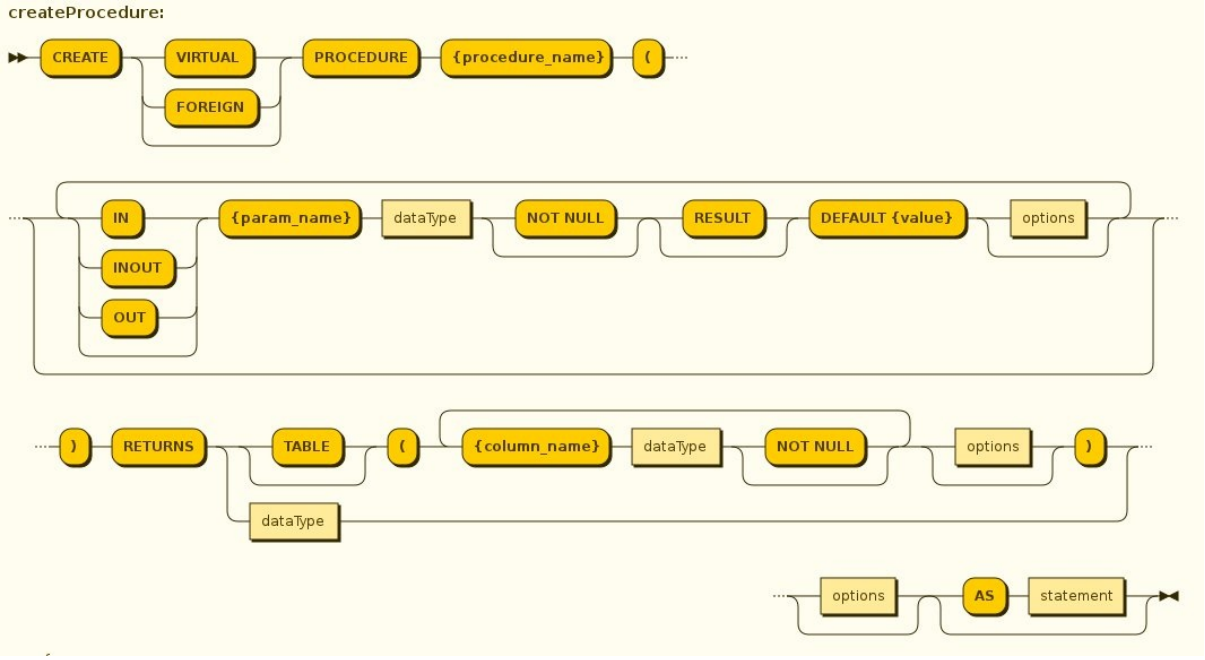
You will typically define a handler for each operation - INSERT/UPDATE/DELTE.

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

Create Procedure/Function

Using the below syntax, user can define a

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



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

Variable Argument Support

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

Example:Vararg procedure

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

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

Property	Data Type or Allowed Values	Description
UUID	string	unique Identifier
NAMEINSOURCE	If this is source function/procedure the name in the physical source, if different from the logical name given above	
ANNOTATION	string	Description of the function/procedure
CATEGORY	string	Function Category
DETERMINISM	NONDETERMINISTIC, COMMAND_DETERMINISTIC, SESSION_DETERMINISTIC, USER_DETERMINISTIC, VDB_DETERMINISTIC, DETERMINISTIC	Not used on virtual procedures
NULL-ON-NULL	'TRUE'|'FALSE'	

JAVA_CLASS	string	Java Class that defines the method in case of UDF
JAVA_METHOD	string	The Java method name on the above defined java class for the UDF implementation
VARARGS	'TRUE'|'FALSE'	Indicates that the last argument of the function can be repeated 0 to any number of times. default false. It is more proper to use a VARIADIC parameter.
AGGREGATE	'TRUE'|'FALSE'	Indicates the function is a user defined aggregate function. Properties specific to aggregates are listed below.

Note that NULL-ON-NULL, VARARGS, and all of the AGGREGATE properties are also valid relational extension metadata properties that can be used on source procedures marked as functions. See also Source Supported Functions for creating FOREIGN functions that are supported by a source.

AGGREGATE FUNCTION OPTIONS:

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

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

Note	**Providing the JAR libraries** - If you have defined a UDF (virtual) function without a Teiid procedure deinition, then it must be accompanied by its implementation in Java. To configure the Java library as dependency to the VDB, see Support for User-Defined Functions

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

Property	Data Type or Allowed Values	Description
UUID	string	Unique Identifier

NAMEINSOURCE	string	In the case of source
ANNOTATION	string	Description of the procedure
UPDATECOUNT	int	if this procedure updates the underlying sources, what is the update count, when update count is >1 the XA protocol for execution is enforced

Example:Define virtual Procedure

```
CREATE VIRTUAL PROCEDURE CustomerActivity(customerid integer)
    RETURNS (name varchar(25), activitydate date, amount decimal)
    AS
    BEGIN
        ...
    END
```

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

Example:Define Virtual Function

```
CREATE VIRTUAL FUNCTION CustomerRank(customerid integer)
   RETURNS integer AS
   BEGIN
   ...
   END
```

Procedure columns may also be marked as NOT NULL, or with a DEFAULT value. Currently only string values are supported as the default value. To have the string interpreted as an expression use the extension property teiid_rel:default_handling set to expression. 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 be nullable and 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.)

Relational Extension OPTIONS:

Property	Data Type or Allowed Values	Description
native-query	Parameterized String	Applies to both functions and procedures. The replacement for the function syntax rather than the standard prefix form with parens. See also Translators#native
non-prepared	boolean	Applies to JDBC procedures using the native-query option. If true a PreparedStatement will not be used to execute the native query.

Example:Native Query

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

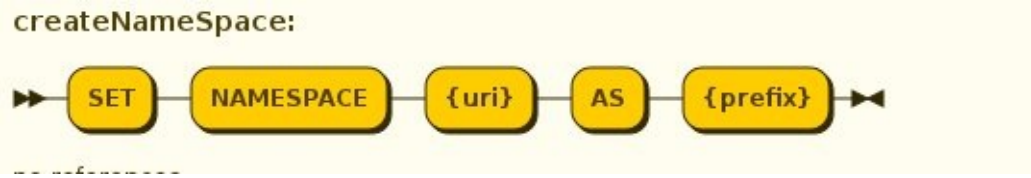
Example:Sequence Native Query

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

Tip	Until Teiid provides higher-level metadata support for sequences, a source function representation is the best fit to expose sequence functionality.

Extension Metadata

When defining the extension metadata in the case of Custom Translators, the properties on tables/views/procedures/columns can define namespace for the properties such that they will not collide with the Teiid specific properties. The property should be prefixed with alias of the Namespace. Prefixes starting with teiid_ are reserved for use by Teiid.



Example of Namespace

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

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

Built-in Namespace Prefixes

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

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 Currently the tooling support for managing the multisource feature is limited, so if you need to use this feature build the VDB as usual in the Teiid Designer and then edit the "vdb.xml" file in the VDB archive using a Text editor to add the additional sources as defined above. 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 in Designer 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.
- If using Designer, you can manually add the multisource column.

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 built by Teiid Designer and supplied to Teiid engine through a VDB archive file. This VDB file contains .INDEX metadata files. By default they are loaded by a *MetadataRepository* with the name *INDEX*. Other 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.

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.

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	GET | POST| PUT | DELETE
URI	URI of procedure	Yes	ex:/procedure
PRODUCES	Type of content produced by the service	no	xml | json | plain | any text
CHARSET	When procedure returns Blob, and content type text based, this character set to used to convert the data	no	US-ASCII | UTF-8

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

Example VDB with REST based metadata properties

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

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

</vdb>
```

Note	<property name="\{http://teiid.org/rest[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.

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>
```

- _auto-generate - will automatically generate the WAR file for the deployed 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 will also implicitly 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. 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);
    }
```

SQL Support

Teiid provides nearly all of the functionality of SQL-92 DML. SQL-99 and later features are constantly being added based upon community need. The following does not attempt to cover SQL exhaustively, but rather highlights SQL's usage within Teiid. See the BNF for SQL Grammar for the exact form of SQL accepted by Teiid.

Identifiers

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

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

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

Syntax Rules:

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

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

- Columns, schemas, and aliases identifiers cannot contain a dot.
- 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.

Reserved Words

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

Expressions

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

Teiid supports the following types of expressions:

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

Column Identifiers

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

Literals

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

Syntax Rules:

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

Some examples of simple literal values are:

```
'abc'
```

escaped single tick

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

```
5
```

scientific notation

```
-37.75e01
```

exact numeric type BigDecimal

```
100.0
```

```
true
```

```
false
```

unicode character

```
'\u0027'
```

binary

```
X'0F0A'
```

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

Date Literal

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

Time Literal

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

Timestamp Literal

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

Or the ANSI keyword syntax:

Date Literal

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

Time Literal

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

Timestamp Literal

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

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

Aggregate Functions

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

Teiid supports the following aggregate functions:

- COUNT(*) – count the number of values (including nulls and duplicates) in a group. Returns an integer - an exception will be thrown if a larger count is computed.
- COUNT(x) – count the number of values (excluding nulls) in a group. Returns an integer - an exception will be thrown if a larger count is computed.
- SUM(x) – sum of the values (excluding nulls) in a group
- AVG(x) – average of the values (excluding nulls) in a group
- MIN(x) – minimum value in a group (excluding null)
- MAX(x) – maximum value in a group (excluding null)
- ANY(x)/SOME(x) – returns TRUE if any value in the group is TRUE (excluding null)
- EVERY(x) – returns TRUE if every value in the group is TRUE (excluding null)
- VAR_POP(x) – biased variance (excluding null) logically equals(sum(x^2) - sum(x)^2/count(x))/count(x); returns a double; null if count = 0
- VAR_SAMP(x) – sample variance (excluding null) logically equals(sum(x^2) - sum(x)^2/count(x))/(count(x) - 1); returns a double; null if count < 2
- STDDEV_POP(x) – standard deviation (excluding null) logically equals SQRT(VAR_POP(x))
- STDDEV_SAMP(x) – sample standard deviation (excluding null) logically equals SQRT(VAR_SAMP(x))
- TEXTAGG(expression [as name], … [DELIMITER char] [QUOTE char | NO QUOTE] [HEADER] [ENCODING id] [ORDER BY …]) – CSV text aggregation of all expressions in each row of a group. When DELIMITER is not specified, by default comma(,) is used as delimiter. All non-null values will be quoted. Double quotes(") is the default quote character. Use QUOTE to specify a different value, or NO QUOTE for no value quoting. If HEADER is specified, the result contains the header row as the first line - the header line will be present even if there are no rows in a group. This aggregation returns a blob.

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

- XMLAGG(xml_expr [ORDER BY …]) – xml concatenation of all xml expressions in a group (excluding null). The ORDER BY clause cannot reference alias names or use positional ordering.
- JSONARRAY_AGG(x [ORDER BY …]) – creates a JSON array result as a Clob including null value. The ORDER BY clause cannot reference alias names or use positional ordering. See also the JSONArray function.

integer value example

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

could return

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

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

string agg example

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

could return

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

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

Syntax Rules:

- Some aggregate functions may contain a keyword 'DISTINCT' before the expression, indicating that duplicate expression values should be ignored. DISTINCT is not allowed in COUNT(*) and is not meaningful in MIN or MAX (result would be unchanged), so it can be used in COUNT, SUM, and AVG.
- Aggregate functions cannot be used in FROM, GROUP BY, or WHERE clauses without an intervening query expression.
- Aggregate functions cannot be nested within another aggregate function without an intervening query expression.
- Aggregate functions may be nested inside other functions.
- Any aggregate function may take an optional FILTER clause of the form

```
FILTER ( WHERE condition )
```

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

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

Window Functions

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

Usage:

```
aggregate|ranking OVER ([PARTITION BY ...]] [ORDER BY ...])
```

aggregate can be any Aggregate Functions. Ranking can be one of ROW_NUMBER(), RANK(), DENSE_RANK().

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.
- The ranking (ROW_NUMBER, RANK, DENSE_RANK) functions require the use of the window specification ORDER BY clause.
- An XMLAGG or JSONARRAY_AGG 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 Function Definitions

- ROW_NUMBER() – functional the same as COUNT(*) with the same window specification. Assigns a number to each row in a partition starting at 1.
- 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.

All values are integers - an exception will be thrown if a larger value is needed.

Processing

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

Teiid will process all window functions with the same window specification together. In general a full pass over the row values coming into the SELECT clause will be required for each unique window specification. For each window specification the values will be grouped according to the PARTITION BY clause. If no PARTITION BY clause is specified, then the entire input is treated as a single partition. The output value is determined based upon the current row value, it’s peers (that is rows that are the same with respect to their ordering), and all prior row values based upon ordering in the partition. The ROW_NUMBER function will assign a unique value to every row regardless of the number of peers.

Example Windowed Results

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

name	salary	max_sal	rank	dense_rank	row_num
John	100000	100000	2	2	2
Henry	50000	50000	5	4	5
John	60000	100000	3	3	3
Suzie	60000	150000	3	3	4
Suzie	150000	150000	1	1	1

Case and Searched Case

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

Supported forms:

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

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

Scalar Subqueries

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

Parameter References

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

Arrays

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

empty arrays

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

single element array

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

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

general array syntax

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

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

An array element reference takes the form of:

```
array_expr[index_expr]
```

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

Operator Precedence

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

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

Criteria

Criteria may be:

- Predicates that evaluate to true or false
- Logical criteria that combines criteria (AND, OR, NOT)
- A value expression with type boolean

Usage:

```
criteria AND|OR criteria
```

```
NOT criteria
```

```
(criteria)
```

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

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

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

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

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

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

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

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

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

LIKE_REGEX allows for standard regular expression syntax to be used for matching. This differs from SIMILAR TO and LIKE in that the escape character is no longer used (\ is already the standard escape mechanism in regular expressions and % and _ have no special meaning. The runtime engine uses the JRE implementation of regular expressions - see the java.util.regex.Pattern class for details.

Note	Teiid does not exhaustively validate LIKE_REGEX pattern values. It is possible to use JRE only regular expression features that are not specified by the SQL specification. Additional not all sources support the same regular expression flavor or extensions. Care should be taken in pushdown situations to ensure that the pattern

	used will have same meaning in Teiid and across all applicable sources.

```
EXISTS (subquery)
```

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

Teiid converts BETWEEN into the equivalent form expression >= minExpression AND expression <= maxExpression

```
expression
```

Where expression has type boolean.

Syntax Rules:

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

Some examples of valid criteria are:

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

Comparing null Values

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

Criteria Precedence

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

Condition	Description
sql operators	See Expressions
EXISTS, LIKE, SIMILAR TO, LIKE_REGEX, BETWEEN, IN, IS NULL, <, <=, >, >=, =, <>	comparison
NOT	negation
AND	conjunction
OR	disjunction

Note however that to prevent lookaheads the parser does not accept all possible criteria sequences. For example "a = b is null" is not accepted, since by the left associative parsing we first recognize "a =", then look for a common value expression. "b is null" is not a valid common value expression. Thus nesting must be used, for example "(a = b) is null". See BNF for SQL Grammar for all parsing rules.

Scalar Functions

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

Numeric Functions

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

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

FORMATINTEGER(x, y)	Formats x using format y	x is integer, y is string, returns string
FORMATLONG(x, y)	Formats x using format y	x is long, y is string, returns string
LOG(x)	Natural log of x (base e)	x in \{double, float}, return type is double
LOG10(x)	Log of x (base 10)	x in \{double, float}, return type is double
MOD(x, y)	Modulus (remainder of x / y)	x in \{integer, long, float, double, biginteger, bigdecimal}, return type is same as x
PARSEBIGDECIMAL(x, y)	Parses x using format y	x, y are strings, returns bigdecimal
PARSEBIGINTEGER(x, y)	Parses x using format y	x, y are strings, returns biginteger
PARSEDOUBLE(x, y)	Parses x using format y	x, y are strings, returns double
PARSEFLOAT(x, y)	Parses x using format y	x, y are strings, returns float
PARSEINTEGER(x, y)	Parses x using format y	x, y are strings, returns integer
PARSELONG(x, y)	Parses x using format y	x, y are strings, returns long
PI()	Value of Pi	return is double
POWER(x,y)	x to the y power	x in \{double, bigdecimal, biginteger}, return is the same type as x
RADIANS(x)	Convert x radians to degrees	x in \{double, bigdecimal}, return type is double
RAND()	Returns a random number, using generator established so far in the query or initializing with system clock if necessary.	Returns double.
RAND(x)	Returns a random number, using new generator seeded with x.	x is integer, returns double.
ROUND(x,y)	Round x to y places; negative values of y indicate places to the left of the decimal point	x in \{integer, float, double, bigdecimal} y is integer, return is same type as x
SIGN(x)	1 if x > 0, 0 if x = 0, -1 if x < 0	x in \{integer, long, float, double, biginteger, bigdecimal}, return type is integer
SIN(x)	Sine value of x	x in \{double, bigdecimal}, return type is double
SQRT(x)	Square root of x	x in \{long, double, bigdecimal}, return type is double

TAN(x)	Tangent of x	x in \{double, bigdecimal}, return type is double
BITAND(x, y)	Bitwise AND of x and y	x, y in {integer}, return type is integer
BITOR(x, y)	Bitwise OR of x and y	x, y in {integer}, return type is integer
BITXOR(x, y)	Bitwise XOR of x and y	x, y in {integer}, return type is integer
BITNOT(x)	Bitwise NOT of x	x in {integer}, return type is integer

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

Parsing Numeric Datatypes from Strings

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

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

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

Formatting Numeric Datatypes as Strings

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

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

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

String Functions

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

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

Function	Definition	Datatype Constraint
x || y	Concatenation operator	x,y in \{string, clob}, return type is string or clob
ASCII(x)	Provide ASCII value of the left most character in x. The empty string will as input will return null.	return type is integer
CHR(x) CHAR(x)	Provide the character for ASCII value x [a]	x in {integer}
CONCAT(x, y)	Concatenates x and y with ANSI semantics. If x and/or y is null, returns null.	x, y in {string}
CONCAT2(x, y)	Concatenates x and y with non-ANSI null semantics. If x and y is null, returns null. If only x or y is null, returns the other value.	x, y in {string}
ENDSWITH(x, y)	Checks if y ends with x. If x or y is null, returns null.	x, y in {string}, returns boolean
INITCAP(x)	Make first letter of each word in string x capital and all others lowercase	x in {string}
INSERT(str1, start, length, str2)	Insert string2 into string1	str1 in {string}, start in {integer}, length in {integer}, str2 in {string}
LCASE(x)	Lowercase of x	x in {string}
LEFT(x, y)	Get left y characters of x	x in {string}, y in {integer}, return string
LENGTH(x)	Length of x	return type is integer
LOCATE(x, y)	Find position of x in y starting at beginning of y	x in {string}, y in {string}, return integer
LOCATE(x, y, z)	Find position of x in y starting at z	x in {string}, y in {string}, z in {integer}, return integer
LPAD(x, y)	Pad input string x with spaces on the left to the length of y	x in {string}, y in {integer}, return string
LPAD(x, y, z)	Pad input string x on the left to the length of y using character z	x in {string}, y in {string}, z in {character}, return string

LTRIM(x)	Left trim x of blank chars	x in {string}, return string
QUERYSTRING(path [, expr [AS name] …])	Returns a properly encoded query string appended to the given path. Null valued expressions are omitted, and a null path is treated as ''. Names are optional for column reference expressions.e.g. `QUERYSTRING('path', 'value' as "&x", ' & ' as y, null as z)` returns `'path?%26x=value&y=%20%26%20'`	path, expr in {string}. name is an identifier
REPEAT(str1,instances)	Repeat string1 a specified number of times	str1 in {string}, instances in {integer} return string
RIGHT(x, y)	Get right y characters of x	x in {string}, y in {integer}, return string
RPAD(input string x, pad length y)	Pad input string x with spaces on the right to the length of y	x in {string}, y in {integer}, return string
RPAD(x, y, z)	Pad input string x on the right to the length of y using character z	x in {string}, y in {string}, z in {character}, return string
RTRIM(x)	Right trim x of blank chars	x is string, return string
SPACE(x)	Repeat the space character x number of times	x is integer, return string
SUBSTRING(x, y) SUBSTRING(x FROM y)	[b] Get substring from x, from position y to the end of x	y in {integer}
SUBSTRING(x, y, z) SUBSTRING(x FROM y FOR z)	[b] Get substring from x from position y with length z	y, z in {integer}
TRANSLATE(x, y, z)	Translate string x by replacing each character in y with the character in z at the same position	x in {string}
TRIM([[LEADING|TRAILING|BOTH] [x] FROM] y)	Trim the leading, trailing, or both ends of a string y of character x. If LEADING/TRAILING/BOTH is not specified, BOTH is used. If no trim character x is specficed then the blank space ' ' is used.	x in {character}, y in {string}
UCASE(x)	Uppercase of x	x in {string}
UNESCAPE(x)	Unescaped version of x. Possible escape sequences are \b - backspace, \t - tab, \n - line feed, \f - form feed, \r - carriage return. \uXXXX, where X is a hex value, can be used to specify any unicode character. \XXX, where X is an octal digit, can be used to specify an octal byte value. If any other character appears after an escape character, that	x in {string}

	character will appear in the output and the escape character will be ignored.	

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

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

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

Some sources however can return an empty string instead of null and some sources do not support negative indexing. If any of these inconsistencies impact you, then please log an issue.

Encoding Functions

TO_CHARS

Return a clob from the blob with the given encoding.

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

BASE64, HEX, 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, and the builtin Java Charset names are valid values for the encoding [b]. x in a clob, encoding is a string, wellformed is a boolean and returns a blob. The two argument form defaults to wellformed=true. If wellformed is false, the conversion function will immediately validate the result such that an unmappable character or malformed input will raise an exception. If wellformed is true, then unmappable characters will be replaced by the default replacement character for the character set. Binary formats, such as BASE64 and HEX, will be checked for correctness regardless of the wellformed parameter.

[b] See the Charset JavaDoc for more on supported Charset names.

Replacement Functions

REPLACE

Replace all occurrences of a given string with another.

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

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

REGEXP_REPLACE

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

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

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

The pattern parameter is expected to be a valid Java Regular Expression

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

flag	name	meaning
g	global	Replace all occurrences, not just the first
m	multiline	Match over multiple lines
i	case insensitive	Match without case sensitivity

Usage:

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

Example regexp_replace

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

Date_Time Functions

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

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

Function	Definition	Datatype Constraint
CURDATE()	Return current date	returns date
CURTIME()	Return current time	returns time
NOW()	Return current timestamp (date and time)	returns timestamp
DAYNAME(x)	Return name of day in the default locale	x in \{date, timestamp}, returns string
DAYOFMONTH(x)	Return day of month	x in \{date, timestamp}, returns integer
DAYOFWEEK(x)	Return day of week (Sunday=1, Saturday=7)	x in \{date, timestamp}, returns integer
DAYOFYEAR(x)	Return day number in year	x in \{date, timestamp}, returns integer
EXTRACT(YEAR|MONTH|DAY|HOUR|MINUTE|SECOND FROM x)	Return the given field value from the date value x. Produces the same result as the assoceated YEAR, MONTH, DAYOFMONTH, HOUR, MINUTE, SECOND functions.The SQL specification also allows for TIMEZONE_HOUR and TIMEZONE_MINUTE as extraction targets. In Teiid all date values are in the timezone of the server.	x in \{date, time, timestamp}, returns integer
FORMATDATE(x, y)	Format date x using format y	x is date, y is string, returns string
FORMATTIME(x, y)	Format time x using format y	x is time, y is string, returns string
FORMATTIMESTAMP(x, y)	Format timestamp x using format y	x is timestamp, y is string, returns string

FROM_UNIXTIME (unix_timestamp)	Return the Unix timestamp (in seconds) as a Timestamp value	Unix timestamp (in seconds)
HOUR(x)	Return hour (in military 24-hour format)	x in \{time, timestamp}, returns integer
MINUTE(x)	Return minute	x in \{time, timestamp}, returns integer
MODIFYTIMEZONE (timestamp, startTimeZone, endTimeZone)	Returns a timestamp based upon the incoming timestamp adjusted for the differential between the start and end time zones. i.e. if the server is in GMT-6, then modifytimezone(\{ts '2006-01-10 04:00:00.0'},'GMT-7', 'GMT-8') will return the timestamp \{ts '2006-01-10 05:00:00.0'} as read in GMT-6. The value has been adjusted 1 hour ahead to compensate for the difference between GMT-7 and GMT-8.	startTimeZone and endTimeZone are strings, returns a timestamp
MODIFYTIMEZONE (timestamp, endTimeZone)	Return a timestamp in the same manner as modifytimezone(timestamp, startTimeZone, endTimeZone), but will assume that the startTimeZone is the same as the server process.	Timestamp is a timestamp; endTimeZone is a string, returns a timestamp
MONTH(x)	Return month	x in \{date, timestamp}, returns integer
MONTHNAME(x)	Return name of month in the default locale	x in \{date, timestamp}, returns string
PARSEDATE(x, y)	Parse date from x using format y	x, y in {string}, returns date
PARSETIME(x, y)	Parse time from x using format y	x, y in {string}, returns time
PARSETIMESTAMP(x,y)	Parse timestamp from x using format y	x, y in {string}, returns timestamp
QUARTER(x)	Return quarter	x in \{date, timestamp}, returns integer
SECOND(x)	Return seconds	x in \{time, timestamp}, returns integer

TIMESTAMPCREATE(date, time)	Create a timestamp from a date and time	date in {date}, time in {time}, returns timestamp
WEEK(x)	Return week in year 1-53, see also System Properties for customization	x in \{date, timestamp}, returns integer
YEAR(x)	Return four-digit year	x in \{date, timestamp}, returns integer

Timestampadd/Timestampdiff

Timestampadd

Add a specified interval amount to the timestamp.

Syntax

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

Arguments

Name	Description
interval	A datetime interval unit, can be one of the following keywords: • SQL_TSI_FRAC_SECOND - fractional seconds (billionths of a second) • SQL_TSI_SECOND - seconds • SQL_TSI_MINUTE - minutes • SQL_TSI_HOUR - hours • SQL_TSI_DAY - days • SQL_TSI_WEEK - weeks using Sunday as the first day • SQL_TSI_MONTH - months • SQL_TSI_QUARTER - quarters (3 months) where the first quarter is months 1-3, etc. • SQL_TSI_YEAR - years
count	An integer represent the datetime need add to timestamp
timestamp	A datetime expression.

Example

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

Timestampdiff

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

Syntax

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

Arguments

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

Example

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

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

Compatibility Issues

- Timestampdiff typically returns an integer, however Teiid's version returns a long. You may receive an exception if you expect a value out of the integer range from a pushed down timestampdiff.
- Teiid's implementation of timestamp diff in 8.2 and prior versions returned values based upon the number of whole canonical interval approximations (365 days in a year, 91 days in a quarter, 30 days in a month, etc.) crossed. For example the difference in months between 2013-03-24 and 2013-04-01 was 0, but based upon the date parts crossed is 1. See System Properties for backwards compatibility.

Parsing Date Datatypes from Strings

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

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

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

Specifying Time Zones

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

For example: GMT-05:00

Type Conversion Functions

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

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

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

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

Choice Functions

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

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

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

Decode Functions

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

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

Within each function call, you include the following arguments:

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

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

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

When the Teiid System encounters the value false in the result set, it replaces the value with 0.

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

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

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

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

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

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

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

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

Lookup Function

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

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

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

Country Code Lookup

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

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

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

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

System Functions

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

Table of Contents

- COMMANDPAYLOAD
- ENV
- NODE_ID
- SESSION_ID
- USER
- CURRENT_DATABASE
- TEIID_SESSION_GET
- TEIID_SESSION_SET

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 environment property.

ENV(key)

To prevent untrusted access to system properties, this function is not enabled by default. The ENV function needs to be enabled via 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>
```

then call using ENV('KEY'), which returns value as string. Ex: ENV('PATH')

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.

XML Functions

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

Table of Contents

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

Sample Data For Examples

Examples provided with XML functions use the following table structure

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

with Data

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

XMLCAST

Cast to or from XML.

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

Expression or type must be XML. The return value will be typed as type. This is the same functionality as XMLTABLE uses to convert values to the desired runtime type - with the exception that array type targets are not supported with XMLCAST.

XMLCOMMENT

Returns an xml comment.

```
XMLCOMMENT(comment)
```

Comment is a string. Return value is xml.

XMLCONCAT

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

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

Content is xml. Return value is xml.

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

Concatenate two or more XML fragments

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

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

XMLELEMENT

Returns an XML element with the given name and content.

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

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

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

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

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

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

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

Simple Example

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

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

Multiple Columns

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

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

Columns as Attributes

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

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

XMLFOREST

Returns an concatenation of XML elements for each content item.

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

See XMLELEMENT for the definition of NSP - XMLNAMESPACES

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

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

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

Example

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

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

XMLAGG

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

```
XMLAGG(xml)
```

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

Example

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


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

XMLPARSE

Returns an XML type representation of the string value expression.

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

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

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

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

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

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

XMLPI

Returns an xml processing instruction.

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

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

XMLQUERY

Returns the XML result from evaluating the given xquery.

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

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

See XMLELEMENT for the definition of NSP - XMLNAMESPACES

Namespaces may also be directly declared in the xquery prolog.

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

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

xquery in string. Return value is xml.

XMLQUERY is part of the SQL/XML 2006 specification.

See also FROM Clause#XMLTABLE

Note	See also XQuery Optimization

XMLEXISTS

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

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

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

See XMLELEMENT for the definition of NSP - XMLNAMESPACES

Namespaces may also be directly declared in the xquery prolog.

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

xquery in string. Return value is boolean.

XMLEXISTS is part of the SQL/XML 2006 specification.

Note	See also XQuery Optimization

XMLSERIALIZE

Returns a character type representation of the xml expression.

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

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

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

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

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

Sample Binary Serialization

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

XMLTEXT

Returns xml text.

```
XMLTEXT(text)
```

text is a string. Return value is xml.

XSLTRANSFORM

Applies an XSL stylesheet to the given document.

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

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

If either argument is null, the result is null.

XPATHVALUE

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

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

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

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

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

Sample XML for xpathValue Ignoring Namespaces

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

Function:

Sample xpathValue Ignoring Namespaces

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

Results in `Hello World`

Examples

Generating hierarchical XML from flat data structure

With following table and its contents

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

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

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

use the SQL statement in Teiid as below

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

another useful link of examples can be found here

JSON Functions

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

Table of Contents

- JSONTOXML
- JSONARRAY
- JSONOBJECT
- JSONPARSE
- Conversion to JSON

JSONTOXML

Returns an xml document from JSON.

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

rootElementName is a string, json is in \{clob, blob}. Return value is xml.

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

The result is always a well-formed XML document.

The mapping to XML uses the following rules:

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

JSON:

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

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

XML:

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

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

JSON:

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

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

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

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

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

JSON:

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

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

XML:

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

```
<?xml version="1.0" ?>
<root>
  <_u002F_invalid>abc</_u002F_invalid>
</root>
```

JSONARRAY

Returns a JSON array.

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

value is any object convertable to a JSON value. Return value is a clob marked as being valid 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"]
```

JSONOBJECT

Returns a JSON object.

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

value is any object convertable to a JSON value. Return value is a clob marked as being valid 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}
```

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 a clob marked as being valid JSON.

A null for either input will return null.

json parse of a simple literal value

```
jsonParse('"a"', true)
```

Conversion to JSON

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

- null values are included as the null literal.
- values parsed as JSON or returned from a JSON construction function (JSONPARSE, JSONARRAY, JSONARRAY_AGG) will be directly appended into a JSON result.
- boolean values are included as true/false literals
- numeric values are included as their default string conversion - in some circumstances if not a number or +-infinity results are allowed, invalid json may be obtained.
- string values are included in their escaped/quoted form.
- binary values are not implicitly convertable to JSON values and require a specific prior to inclusion in JSON.
- all other values will be included as their string conversion in the appropriate escaped/quoted form.

Security Functions

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

Table of Contents

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

HASROLE

Whether the current caller has the Teiid data role roleName.

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

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

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

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

MD5

Computes the MD5 hash of the value.

```
MD5(value)
```

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

SHA1

Computes the SHA-1 hash of the value.

```
SHA1(value)
```

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

SHA2_256

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

```
SHA2_256(value)
```

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

SHA2_512

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

```
SHA2_512(value)
```

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

AES_ENCRYPT

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

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

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

AES_DECRYPT

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

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

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

Spatial Functions

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

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

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

Table of Contents

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

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

Conversion Functions

ST_GeomFromText

Returns a geometry from a Clob in WKT format.

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

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

ST_GeomFromWKB/ST_GeomFromBinary

Returns a geometry from a blob in WKB format.

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

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

ST_GeomFromEWKB

Returns a geometry from a blob in EWKB format.

```
ST_GeomFromEWKB(bin)
```

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

ST_GeomFromText

Returns a geometry from a Clob in EWKT format.

```
ST_GeomFromEWKT(text)
```

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

ST_GeomFromGeoJSON

Returns a geometry from a Clob in GeoJSON format.

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

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

ST_GeomFromGML

Returns a geometry from a Clob in GML2 format.

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

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

ST_AsText

```
ST_AsText(geom)
```

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

ST_AsBinary

```
ST_AsBinary(geom)
```

geom is a geometry. Return value is a blob in WKB format.

ST_AsEWKB

```
ST_AsEWKB(geom)
```

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

ST_AsGeoJSON

```
ST_AsGeoJSON(geom)
```

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

ST_AsGML

```
ST_AsGML(geom)
```

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

ST_AsEWKT

```
ST_AsEWKT(geom)
```

geom is a geometry. Return value is a clob with the EWKT value. The EWKT value is the WKT value with the SRID prefix.

ST_AsKML

```
ST_AsKML(geom)
```

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

Operators

&&

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

```
geom1 && geom2
```

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

Relationship Functions

ST_Contains

Returns true if geom1 contains geom2 contains another.

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

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

ST_Crosses

Returns true if the geometries cross.

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

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

ST_Disjoint

Returns true if the geometries are disjoint.

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

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

ST_Distance

Returns the distance between two geometries.

```
ST_Distance(geom1, geom2)
```

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

ST_DWithin

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

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

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

ST_Equals

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

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

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

ST_Intersects

Returns true if the geometries intersect.

```
ST_Intersects(geom1, geom2)
```

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

ST_OrderingEquals

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

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

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

ST_Overlaps

Returns true if the geometries overlap.

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

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

ST_Relate

Test or return the intersection of geom1 and geom2.

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

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

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

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

ST_Touches

Returns true if the geometries touch.

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

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

ST_Within

Returns true if geom1 is completely inside geom2.

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

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

Attributes and Tests

ST_Area

Returns the area of geom.

```
ST_Area(geom)
```

geom is a geometry. Return value is a double.

ST_CoordDim

Returns the coordinate dimensions of geom.

```
ST_CoordDim(geom)
```

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

ST_Dimension

Returns the dimension of geom.

```
ST_Dimension(geom)
```

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

ST_EndPoint

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

```
ST_EndPoint(geom)
```

geom is a geometry. Return value is a geometry.

ST_ExteriorRing

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

```
ST_ExteriorRing(geom)
```

geom is a geometry. Return value is a geometry.

ST_GeometryN

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

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

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

ST_GeometryType

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

```
ST_GeometryType(geom)
```

geom is a geometry. Return value is a string.

ST_HasArc

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

```
ST_HasArc(geom)
```

geom is a geometry. Return value is a geometry.

ST_InteriorRingN

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

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

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

ST_IsClosed

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

```
ST_IsClosed(geom)
```

geom is a geometry. Return value is a boolean.

ST_IsEmpty

Returns true if the set of points is empty.

```
ST_IsEmpty(geom)
```

geom is a geometry. Return value is a boolean.

ST_IsRing

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

```
ST_IsRing(geom)
```

geom is a geometry. Return value is a boolean.

ST_IsSimple

Returns true if the geom is simple.

```
ST_IsSimple(geom)
```

geom is a geometry. Return value is a boolean.

ST_IsValid

Returns true if the geom is valid.

```
ST_IsValid(geom)
```

geom is a geometry. Return value is a boolean.

ST_Length

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

```
ST_Length(geom)
```

geom is a geometry. Return value is a double.

ST_NumGeometries

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

```
ST_NumGeometries(geom)
```

geom is a geometry. Return value is an integer.

ST_NumInteriorRings

Returns the number of interior rings in the Polygon geom. Returns null if geom is not a Polygon.

```
ST_NumInteriorRings(geom)
```

geom is a geometry. Return value is an integer.

ST_NunPoints

Returns the number of Points in geom.

```
ST_NunPoints(geom)
```

geom is a geometry. Return value is an integer.

ST_PointOnSurface

Returns a Point that is guarenteed to be on the surface of geom.

```
ST_PointOnSurface(geom)
```

geom is a geometry. Return value is a Point geometry.

ST_Perimeter

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

```
ST_Perimeter(geom)
```

geom is a geometry. Return value is a double.

ST_PointN

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

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

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

ST_SRID

Returns the SRID for the geometry.

```
ST_SRID(geom)
```

geom is a geometry. 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(geom, srid)
```

geom is a geometry. srid is an integer. Return value is a geometry. Only the SRID metadata of the geometry is modified.

ST_StartPoint

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

```
ST_StartPoint(geom)
```

geom is a geometry. Return value is a geometry.

ST_X

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

```
ST_X(geom)
```

geom is a geometry. Return value is a double.

ST_Y

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

```
ST_Y(geom)
```

geom is a geometry. Return value is a double.

ST_Z

Returns the Z ordinate value, or null if the Point is empty. Throws an exception if the Geometry is not a Point. Will typically return null as 3 dimensions are not fully supported.

```
ST_Z(geom)
```

geom is a geometry. Return value is a double.

Misc. Functions

ST_Boundary

Computes the boundary of the given geometry.

```
ST_Boundary(geom)
```

geom is a geometry. Return value is a geometry.

ST_Buffer

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

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

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

ST_Centroid

Computes the geometric center Point of geom.

```
ST_Centroid(geom)
```

geom is a geometry. Return value is a geometry.

ST_ConvexHull

Return the smallest convex Polygon that contains all of the points in geom.

```
ST_ConvexHull(geom)
```

geom is a geometry. Return value is a geometry.

ST_CurveToLine

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

```
ST_CurveToLine(geom)
```

geom is a geometry. Return value is a geometry.

ST_Difference

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

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

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

ST_Envelope

Computes the 2D bounding box of the given geometry.

```
ST_Envelope(geom)
```

geom is a geometry. Return value is a geometry.

ST_Force_2D

Removes the z coordinate value if present.

```
ST_Force_2D(geom)
```

geom is a geometry. Return value is a geometry.

ST_Intersection

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

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

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

ST_Simplify

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

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

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

ST_SimplifyPreserveTopology

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

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

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

ST_SnapToGrid

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

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

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

ST_SymDifference

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

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

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

ST_Transform

Transforms the geometry value from one coordinate system to another.

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

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

ST_Union

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

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

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

Aggregate Functions

ST_Extent

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

```
ST_Extent(geom)
```

geom is a geometry. Return value is a geometry.

Construction Functions

ST_Point

Retuns the Point for the given cooridinates.

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

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

ST_Polygon

Retuns the Polygon with the given shell and srid.

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

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

Miscellaneous Functions

Documents additional functions and those contributed by other projects.

Table of Contents

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

Array functions

array_get

Returns the object value at a given array index.

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

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

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

array_length

Returns the length for a given array

```
array_length(array)
```

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

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

Other Functions

uuid

Retuns a universally unique identifier.

```
uuid()
```

the return type is string.

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

Data Quality Functions

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

osdq.random

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

```
random(sourceValue)
```

The sourceValue is the string that need to randomize.

osdq.digit

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

```
digit(sourceValue)
```

The sourceValue is the string that need to digit.

osdq.whitespaceIndex

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

```
whitespaceIndex(sourceValue)
```

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

osdq.validCreditCard

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

```
validCreditCard(cc)
```

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

osdq.validSSN

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

```
validSSN(ssn)
```

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

osdq.validPhone

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

```
validPhone(phone)
```

The phone is the phone number string need to check.

osdq.validEmail

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

```
validEmail(email)
```

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

osdq.cosineDistance

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

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

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

osdq.jaccardDistance

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

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

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

osdq.jaroWinklerDistance

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

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

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

osdq.levenshteinDistance

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

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

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

Nondeterministic Function Handling

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

1. **Deterministic** - the function will always return the same result for the given inputs. Deterministic functions are evaluated by the engine as soon as all input values are known, which may occur as soon as the rewrite phase. Some functions, such as the lookup function, are not truly deterministic, but is treated as such for performance. All functions not categorized below are considered deterministic.
2. **User Deterministic** - the function will return the same result for the given inputs for the same user. This includes the hasRole and user functions. User deterministic functions are evaluated by the engine as soon as all input values are known, which may occur as soon as the rewrite phase. If a user deterministic function is evaluated during the creation of a prepared processing plan, then the resulting plan will be cached only for the user.
3. **Session Deterministic** - the function will return the same result for the given inputs under the same user session. This category includes the env function. Session deterministic functions are evaluated by the engine as soon as all input values are known, which may occur as soon as the rewrite phase. If a session deterministic function is evaluated during the creation of a prepared processing plan, then the resulting plan will be cached only for the user’s session.
4. **Command Deterministic** - the result of function evaluation is only deterministic within the scope of the user command. This category include the curdate, curtime, now, and commandpayload functions. Command deterministic functions are delayed in evaluation until processing to ensure that even prepared plans utilizing these functions will be executed with relevant values. Command deterministic function evaluation will occur prior to pushdown - however multiple occurrences of the same command deterministic time function are not guaranteed to evaluate to the same value.
5. **Nondeterministic** - the result of function evaluation is fully nondeterministic. This category includes the rand function and UDFs marked as nondeterministic. Nondeterministic functions are delayed in evaluation until processing with a preference for pushdown. If the function is not pushed down, then it may be evaluated for every row in it’s execution context (for example if the function is used in the select clause).

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

DML Commands

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

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

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

SELECT Command

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

A SELECT command has a number of clauses:

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

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

- WITH stage - gathers all rows from all with items in the order listed. Subsequent with items and the main query can reference the a with item as if it is a table.
- FROM stage - gathers all rows from all tables involved in the query and logically joins them with a Cartesian product, producing a single large table with all columns from all tables. Joins and join criteria are then applied to filter rows that do not match the join structure.
- WHERE stage - applies a criteria to every output row from the FROM stage, further reducing the number of rows.
- GROUP BY stage - groups sets of rows with matching values in the group by columns.
- HAVING stage - applies criteria to each group of rows. Criteria can only be applied to columns that will have constant values within a group(those in the grouping columns or aggregate functions applied across the group).

- SELECT stage - specifies the column expressions that should be returned from the query. Expressions are evaluated, including aggregate functions based on the groups of rows, which will no longer exist after this point. The output columns are named using either column aliases or an implicit name determined by the engine. If SELECT DISTINCT is specified, duplicate removal will be performed on the rows being returned from the SELECT stage.
- ORDER BY stage - sorts the rows returned from the SELECT stage as desired. Supports sorting on multiple columns in specified order, ascending or descending. The output columns will be identical to those columns returned from the SELECT stage and will have the same name.
- LIMIT stage - returns only the specified rows (with skip and limit values).

This model can be used to understand many questions about SQL. For example, columns aliased in the SELECT clause can only be referenced by alias in the ORDER BY clause. Without knowledge of the processing model, this can be somewhat confusing. Seen in light of the model, it is clear that the ORDER BY stage is the only stage occurring after the SELECT stage, which is where the columns are named. Because the WHERE clause is processed before the SELECT, the columns have not yet been named and the aliases are not yet known.

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

VALUES Command

The VALUES command is used to construct a simple table.

Example Syntax

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

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

A VALUES command with a single value set is equivalent to "SELECT value, ….". A VALUES command with multiple values sets is equivalent to a UNION ALL of simple SELECTs - "SELECT value, …. UNION ALL SELECT valueX, …".

Update Commands

Update commands can report integer update counts. If a larger number or rows is updated, then the max integer value will be reported (2^31 -1).

INSERT Command

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

Example Syntax

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

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

UPDATE Command

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

Example Syntax

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

DELETE Command

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

Example Syntax

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

UPSERT/MERGE Command

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

Example Syntax

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

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

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

EXECUTE Command

The EXECUTE command is used to execute a procedure, such as a virtual procedure or a stored procedure. Procedures may have zero or more scalar input parameters. The return value from a procedure is a result set or the set of inout/out/return scalars. Note that EXEC or CALL can be used as a short form of this command.

Example Syntax

```
EXECUTE proc()
```

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

Named Parameter Syntax

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

Syntax Rules:

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

Procedural Relational Command

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

Example Syntax

```
select * from proc
```

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

```
select output_param1, output_param2 from proc, table where input_param1 = table.col1 and input_param2 = table.c
ol2
```

Syntax Rules:

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

Multiple Execution

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

Alternative Syntax

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

Anonymous Procedure Block

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

Example Syntax

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

Syntax Rules:

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

Set Operations

Teiid supports the UNION, UNION ALL, INTERSECT, EXCEPT set operation as a way of combining the results of query expressions.

Usage:

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

Syntax Rules:

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

Subqueries

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

Supported subquery types:

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

Inline views

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

Example Subquery in FROM Clause (Inline View)

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

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

Subqueries are supported in quantified criteria, the EXISTS predicate, the IN predicate, and as Scalar Subqueries.

Example Subquery in WHERE Using EXISTS

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

Example Quantified Comparison Subqueries

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

Example IN Subquery

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

See also Subquery Optimization.

WITH Clause

Teiid supports non-recursive common table expressions via the WITH clause. WITH clause items may be referenced as tables in subsequent with clause items and in the main query. The WITH clause can be thought of as providing query scoped temporary tables.

Usage:

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

Syntax Rules:

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

Note	The WITH clause is also subject to optimization and it’s entries may not be processed if they are not needed in the subsequent query.

Examples:

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

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

Recursive Common Table Expressions

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

Usage:

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

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

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

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

been exceeded an exception will be thrown.

Example:

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

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

SELECT Clause

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

Usage:

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

Syntax Rules:

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

FROM Clause

The FROM clause specifies the target table(s) for SELECT, UPDATE, and DELETE statements.

Example Syntax:

- FROM table [[AS] alias]
- FROM table1 [INNER|LEFT OUTER|RIGHT OUTER|FULL OUTER] JOIN table2 ON join-criteria
- FROM table1 CROSS JOIN table2
- FROM (subquery) [AS] alias
- FROM TABLE(subquery) [AS] alias
- FROM table1 JOIN /*+ MAKEDEP */ table2 ON join-criteria
- FROM table1 JOIN /*+ MAKENOTDEP */ table2 ON join-criteria
- FROM /*+ MAKEIND */ table1 JOIN table2 ON join-criteria
- FROM /*+ NO_UNNEST */ vw1 JOIN table2 ON join-criteria
- FROM table1 left outer join /*+ optional */ table2 ON join-criteria
- FROM TEXTTABLE…
- FROM XMLTABLE…
- FROM ARRAYTABLE…
- FROM OBJECTTABLE…
- FROM (SELECT …

From Clause Hints

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

Example Hint

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

Dependent Joins

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

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

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

MAKEDEP and MAKEIND support optional max and join arguments:

- MAKEDEP(JOIN) means that the entire join should be pushed
- MAKEDEP(NO JOIN) means that the entire join should not be pushed
- MAKEDEP(MAX:val) meaning that the dependent join should only be performed if there are less than the max number of values from the independent side.

Other Hints

NO_UNNEST can be specified against a subquery from clause or view to instruct the planner to not merge the nested SQL in the surrounding query - also known as view flattening. This hint only applies to Teiid planning and is not passed to source queries. NO_UNNEST may appear in a comment that proceeds the from clause.

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

Example PRESERVE Hint

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

Nested Table Reference

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

A nested table may have correlated references to preceding FROM clause column references as long as INNER and LEFT OUTER joins are used. This is especially useful in cases where then nested expression is a procedure or function call.

Valid example:

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

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

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

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

XMLTABLE

The XMLTABLE function uses XQuery to produce tabular output. The XMLTABLE function is implicitly a nested table and may be correlated to preceding FROM clause entries. XMLTABLE is part of the SQL/XML 2006 specification.

Usage:

```
XMLTABLE([<NSP>,] xquery-expression [<PASSING>] [COLUMNS <COLUMN>, ... )] AS name
```

```
COLUMN := name (FOR ORDINALITY | (datatype [DEFAULT expression] [PATH string]))
```

See XMLELEMENT for the definition of NSP - XMLNAMESPACES.

See XMLQUERY for the definition of PASSING.

See also XMLQUERY

Note	See also XQuery Optimization

Parameters

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

Syntax Rules:

- Only 1 FOR ORDINALITY column may be specified.
- The columns names must not contain duplicates.
- The blob datatype is supported, but there is only built-in support for xs:hexBinary values. For xs:base64Binary, use a workaround of a PATH that uses the explicit value constructor "xs:base64Binary(<path>)".
- The column expression must evaluate to a single value if a non-array type is expected.
- If an array type is specified then an array will be returned unless there are no elements in the sequence, in which case a null value is returned.
- An empty element is not a valid null value as it is effectively the empty string. The xsi:nil attribute should be used to define convey a null valued element.

Examples

Use of passing, returns 1 row [1]:

```
select * from xmltable('/a' PASSING xmlparse(document '<a id="1"/>') COLUMNS id integer PATH '@id') x
```

As a nested table:

```
select x.* from t, xmltable('/x/y' PASSING t.doc COLUMNS first string, second FOR ORDINALITY) x
```

Invalid multi-value:

```
select * from xmltable('/a' PASSING xmlparse(document '<a><b id="1"/><b id="2"/></a>') COLUMNS id integer PATH
'b/@id') x
```

Array multi-value:

```
select * from xmltable('/a' PASSING xmlparse(document '<a><b id="1"/><b id="2"/></a>') COLUMNS id integer[] PATH
 'b/@id') x
```

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

```
select * from xmltable('/a' PASSING xmlparse(document '<a xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
><b xsi:nil="true"/></a>') COLUMNS id integer PATH 'b') x
```

ARRAYTABLE

The ARRAYTABLE function processes an array input to produce tabular output. The function itself defines what columns it projects. The ARRAYTABLE function is implicitly a nested table and may be correlated to preceding FROM clause entries.

Usage:

```
ARRAYTABLE(expression COLUMNS <COLUMN>, ...) AS name
COLUMN := name datatype
```

Parameters

- expression - the array to process, which should be a java.sql.Array or java array value.

Syntax Rules:

- The columns names must be not contain duplicates.

Examples

- As a nested table:

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

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

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

is the same as

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


- *Prev*FROM Clause
- **Top of page**
- **Front page**
- *Next*OBJECTTABLE

OBJECTTABLE

The OBJECTTABLE function processes an object input to produce tabular output. The function itself defines what columns it projects. The OBJECTTABLE function is implicitly a nested table and may be correlated to preceding FROM clause entries.

Usage:

```
OBJECTTABLE([LANGUAGE lang] rowScript [PASSING val AS name ...] COLUMNS colName colType colScript [DEFAULT defa
ultExpr] ...) AS id
```

Parameters

- lang - an optional string literal that is the case sensitive language name of the scripts to be processed. The script engine must be available via a JSR-223 ScriptEngineManager lookup. In some instances this may mean making additional modules available to your vdb, which can be done via the same process as adding modules/libraries for UDFs. If a LANGUAGE is not specified, the default of 'teiid_script' (see below) will be used.
- name - an identifier that will bind the val expression value into the script context.
- rowScript is a string literal specifying the script to create the row values. for each non-null item the Iterator produces the columns will be evaluated.
- colName/colType are the id/data type of the column, which can optionally be defaulted with the DEFAULT clause expression defaultExpr.
- colScript is a string literal specifying the script that evaluates to the column value.

Syntax Rules:

- The columns names must be not contain duplicates.
- Teiid will place several special variables in the script execution context. The CommandContext is available as teiid_context. Additionally the colScripts may access teiid_row and teiid_row_number. teiid_row is the current row object produced by the row script. teiid_row_number is the current 1-based row number.
- rowScript is evaluated to an Iterator. If the results is already an Iterator, it is used directly. If the evaluation result is an Iteratable, Array, or Array type, then an Iterator will be obtained. Any other Object will be treated as an Iterator of a single item). In all cases null row values will be skipped.

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

Examples

- Accessing special variables:

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

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

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

Data roles are also secured with Data Roles using the language permission.

teiid_script

teiid_script is a simple scripting expression language that allows access to passing and special variables as well as any non-void 0-argument methods on objects and indexed values on arrays/lists. A teiid_script expression begins by referencing the passing or special variable. Then any number of `.' accessors may be chained to evaluate the expression to a different value. Methods may be accessed by their property names, for example foo rather than getFoo. If the object both a `getFoo()` and `foo()` method, then the accessor foo references `fo ()` and getFoo should be used to call the getter. An array or list index may be accessed using a 1-based positive integral value - using the same '.' accessor syntax. The same logic as the system function array_get is used meaning that null will be returned rather than exception if the index is out of bounds.

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

Examples

- To get the VDB description string:

```
teiid_context.session.vdb.description
```

- To get the first character of the VDB description string:

```
teiid_context.session.vdb.description.toCharArray.1
```

TEXTTABLE

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

Usage:

```
TEXTTABLE(expression [SELECTOR string] COLUMNS <COLUMN>, ... [NO ROW DELIMITER | ROW DELIMITER char] [DELIMITER
 char] [(QUOTE|ESCAPE) char] [HEADER [integer]] [SKIP integer] [NO TRIM]) AS name
```

Where <COLUMN>

```
COLUMN := name (FOR ORDINALITY | ([HEADER string] datatype [WIDTH integer [NO TRIM]] [SELECTOR string integer]))
```

Parameters

- expression - the text content to process, which should be convertible to CLOB.
- SELECTOR is used with files containing multiple types of rows (example: order header, detail, summary). A TEXTTABLE SELECTOR specifies which lines to include in the output. Matching lines must begin with the selector string. The selector in column delimited files must be followed by the column delimiter.
    - If a TEXTTABLE SELECTOR is specified, a SELECTOR may also be specified for column values. A column SELECTOR argument will select the nearest preceding text line with the given SELECTOR prefix and select the value at the given 1-based integer position (which includes the selector itself). If no such text line or position with a given line exists, a null value will be produced. A column SELECTOR is not valid with fixed width parsing.
- NO ROW DELIMITER indicates that fixed parsing should not assume the presence of newline row delimiters.
- ROW DELIMITER sets the row delimiter / new line to an alternate character. Defaults to the new line character - with built in handling for treating carriage return new line as a single character. If ROW DELIMITER is specified, carriage return will be given no special treatment.
- DELIMITER sets the field delimiter character to use. Defaults to ','.
- QUOTE sets the quote, or qualifier, character used to wrap field values. Defaults to "".
- ESCAPE sets the escape character to use if no quoting character is in use. This is used in situations where the delimiter or new line characters are escaped with a preceding character, e.g. \,
- HEADER specifies the text line number (counting every new line) on which the column names occur. If the HEADER option for a column is specified, then that will be used as the expected header name. All lines prior to the header will be skipped. If HEADER is specified, then the header line will be used to determine the TEXTTABLE column position by case-insensitive name matching. This is especially useful in situations where only a subset of the columns are needed. If the HEADER value is not specified, it defaults to 1. If HEADER is not specified, then columns are expected to match positionally with the text contents.
- SKIP specifies the number of text lines (counting every new line) to skip before parsing the contents. HEADER may still be specified with SKIP.
- A FOR ORDINALITY column is typed as integer and will return the 1-based item number as its value.

- WIDTH indicates the fixed-width length of a column in characters - not bytes. With the default ROW DELIMITER, a CR NL sequence counts as a single character.
- NO TRIM specified on the TEXTTABLE, it will affect all column and header values. If NO TRIM is specified on a column, then the fixed or unqualified text value not be trimmed of leading and trailing white space.

Syntax Rules:

- If width is specified for one column it must be specified for all columns and be a non-negative integer.
- If width is specified, then fixed width parsing is used ESCAPE, QUOTE, column SELECTOR, nor HEADER should not be specified.
- If width is not specified, then NO ROW DELIMITER cannot be used.
- The columns names must not contain duplicates.
- The QUOTE, DELIMITER, and ROW DELIMITER must all be different characters.

Examples

- Use of the HEADER parameter, returns 1 row ['b']:

```
SELECT * FROM TEXTTABLE(UNESCAPE('col1,col2,col3\na,b,c') COLUMNS col2 string HEADER) x
```

- Use of fixed width, returns 2 rows ['a', 'b', 'c'], ['d', 'e', 'f']:

```
SELECT * FROM TEXTTABLE(UNESCAPE('abc\ndef') COLUMNS col1 string width 1, col2 string width 1, col3 string widt
h 1) x
```

- Use of fixed width without a row delimiter, returns 3 rows ['a'], ['b'], ['c']:

```
SELECT * FROM TEXTTABLE('abc' COLUMNS col1 string width 1 NO ROW DELIMITER) x
```

- Use of ESCAPE parameter, returns 1 row ['a,', 'b']:

```
SELECT * FROM TEXTTABLE('a:,,b' COLUMNS col1 string, col2 string ESCAPE ':') x
```

- As a nested table:

```
SELECT x.* FROM t, TEXTTABLE(t.clobcolumn COLUMNS first string, second date SKIP 1) x
```

- Use of SELECTORs, returns 2 rows ['c', 'd', 'b'], ['c', 'f', 'b']:

```
SELECT * FROM TEXTTABLE('a,b\nc,d\nc,f' SELECTOR 'c' COLUMNS col1 string, col2 string col3 string SELECTOR 'a' 2
) x
```

WHERE Clause

The WHERE clause defines the criteria to limit the records affected by SELECT, UPDATE, and DELETE statements.

The general form of the WHERE is:

- WHERE Criteria

GROUP BY Clause

The GROUP BY clause denotes that rows should be grouped according to the specified expression values. One row will be returned for each group, after optionally filtering those aggregate rows based on a HAVING clause.

The general form of the GROUP BY is:

```
GROUP BY expression [,expression]*
```

```
GROUP BY ROLLUP(expression [,expression]*)
```

Syntax Rules:

- Column references in the group by cannot be made to alias names in the SELECT clause.
- Expressions used in the group by must appear in the select clause.
- Column references and expressions in the SELECT/HAVING/ORDER BY clauses that are not used in the group by clause must appear in aggregate functions.
- If an aggregate function is used in the SELECT clause and no GROUP BY is specified, an implicit GROUP BY will be performed with the entire result set as a single group. In this case, every column in the SELECT must be an aggregate function as no other column value will be fixed across the entire group.
- The group by columns must be of a comparable type.

Rollups

Just like normal grouping, rollup processing logically occurs before the HAVING clause is processed. A ROLLUP of expressions will produce the same output as a regular grouping with the addition of aggregate values computed at higher aggregation levels. For N expressions in the ROLLUP, aggregates will be provided over (), (expr1), (expr1, expr2), etc. up to (expr1, … exprN-1) with the other grouping expressions in the output as null values. For example with the normal aggregation query

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

returning:

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

The rollup query

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

would return:

country	city	sum(amount)
US	St. Louis	10000
US	Raleigh	150000
US	Denver	20000
US	**<null>**	180000
UK	Birmingham	50000
UK	London	75000
UK	**<null>**	125000
<null>	**<null>**	305000

Note	Not all sources support ROLLUPs and some optimizations compared to normal aggregate processing may be inhibited by the use of a ROLLUP.

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

HAVING Clause

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

Syntax Rules:

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

ORDER BY Clause

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

Usage:

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

Syntax Rules:

- Sort columns may be specified positionally by a 1-based positional integer, by SELECT clause alias name, by SELECT clause expression, or by an unrelated expression.
- Column references may appear in the SELECT clause as the expression for an aliased column or may reference columns from tables in the FROM clause. If the column reference is not in the SELECT clause the query must not be a set operation, specify SELECT DISTINCT, or contain a GROUP BY clause.
- Unrelated expressions, expressions not appearing as an aliased expression in the select clause, are allowed in the order by clause of a non-set QUERY. The columns referenced in the expression must come from the from clause table references. The column references cannot be to alias names or positional.
- The ORDER BY columns must be of a comparable type.
- If an ORDER BY is used in an inline view or view definition without a limit clause, it will be removed by the Teiid optimizer.
- If NULLS FIRST/LAST is specified, then nulls are guaranteed to be sorted either first or last. If the null ordering is not specified, then results will typically be sorted with nulls as low values, which is Teiid’s internal default sorting behavior. However not all sources return results with nulss sorted as low values by default, and Teiid may return results with different null orderings.

Warning	The use of positional ordering is no longer supported by the ANSI SQL standard and is a deprecated feature in Teiid. It is preferable to use alias names in the order by clause.

LIMIT Clause

The LIMIT clause specifies a limit on the number of records returned from the SELECT command. An optional offset (the number of rows to skip) can be specified. The LIMIT clause can also be specified using the SQL 2008 OFFSET/FETCH FIRST clauses. If an ORDER BY is also specified, it will be applied before the OFFSET/LIMIT are applied. If an ORDER BY is not specified there is generally no guarantee what subset of rows will be returned.

Usage:

```
LIMIT [offset,] limit
```

```
[OFFSET offset ROW|ROWS] [FETCH FIRST|NEXT [limit] ROW|ROWS ONLY]
```

Syntax Rules:

- The limit/offset expressions must be a non-negative integer or a parameter reference (?). An offset of 0 is ignored. A limit of 0 will return no rows.
- The terms FIRST/NEXT are interchangeable as well as ROW/ROWS.
- The limit clause may take an optional preceding NON_STRICT hint to indicate that push operations should not be inhibited even if the results will not be consistent with the logical application of the limit. The hint is only needed on unordered limits, e.g. "SELECT * FROM VW /*+ NON_STRICT */ LIMIT 2".

Examples:

- LIMIT 100 - returns the first 100 records (rows 1-100)
- LIMIT 500, 100 - skips 500 records and returns the next 100 records(rows 501-600)
- OFFSET 500 ROWS - skips 500 records
- OFFSET 500 ROWS FETCH NEXT 100 ROWS ONLY - skips 500 records and returns the next 100 records (rows 501-600)
- FETCH FIRST ROW ONLY - returns only the first record

INTO Clause

Warning	Usage of the INTO Clause for inserting into a table has been been deprecated. An INSERT with a query command should be used instead.

When the into clause is specified with a SELECT, the results of the query are inserted into the specified table. This is often used to insert records into a temporary table. The INTO clause immediately precedes the FROM clause.

Usage:

```
INTO table FROM ...
```

Syntax Rules:

- The INTO clause is logically applied last in processing, after the ORDER BY and LIMIT clauses.
- Teiid’s support for SELECT INTO is similar to MS SQL Server. The target of the INTO clause is a table where the result of the rest select command will be inserted. SELECT INTO should not be used UNION query.

OPTION Clause

The OPTION keyword denotes options the user can pass in with the command. These options are Teiid specific and not covered by any SQL specification.

Usage:

```
OPTION option (, option)*
```

Supported options:

- MAKEDEP table (,table)* - specifies source tables that should be made dependent in the join
- MAKEIND table (,table)* - specifies source tables that should be made dependent in the join
- MAKENOTDEP table (,table)* - prevents a dependent join from being used
- NOCACHE [table (,table)*] - prevents cache from being used for all tables or for the given tables

Examples:

- OPTION MAKEDEP table1
- OPTION NOCACHE

All tables specified in the OPTION clause should be fully qualified, however the name may match either an alias name or the fully qualified name.

The makedep and makeind hints can take optional arguments to control the dependent join. The extended hint form is:

```
MAKEDEP tbl([max:val] [[no] join])
```

- tbl(JOIN) means that the entire join should be pushed
- tbl(NO JOIN) means that the entire join should not be pushed
- tbl(MAX:val) meaning that the dependent join should only be performed if there are less than the max number of values from the independent side.

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

Note	MAKEDEP and MAKENOTDEP hints may take table names in the form of @view1.view2…table. For example with an inline view "select * from (select * from tbl1, tbl2 where tbl1.c1 = tbl2.c2) as v1 option makedep @v1.tbl1" the hint will now be understood as applying under the v1 view.

DDL Commands

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

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

Temp Tables

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

Table of Contents

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

Local Temporary Tables

Local temporary tables can be defined implicitly by referencing them in a INSERT statement or explicitly with a CREATE TABLE statement. Implicitly created temp tables must have a name that starts with `#`.

Explicit Creation syntax

```
CREATE LOCAL TEMPORARY TABLE name (column type [NOT NULL], ... [PRIMARY KEY (column, ...)]) [ON COMMIT PRESERVE ROWS]
```

- Use the SERIAL data type to specify a NOT NULL and auto-incrementing INTEGER column. The starting value of a SERIAL column is 1.

Implicit Creation syntax

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

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

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

Drop syntax

```
DROP TABLE name
```

Example

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

```
CREATE LOCAL TEMPORARY TABLE TEMP (a integer, b integer, c integer);
SELECT * INTO temp FROM Src1;
SELECT * INTO temp FROM Src2;
INSERT INTO temp VALUES (1,2,3);
```

```
SELECT a,b,c FROM Src3, temp WHERE Src3.a = temp.b;
```

See Virtual Procedures for more on local temporary table usage.

Global Temporary Tables

Global temporary tables are created in Teiid Designer or via the metadata supplied to Teiid at deploy time. Unlike local temporary tables, they cannot be created at runtime. A global temporary tables share a common definition via a schema entry, but each session has a new instance of the temporary table created upon it's first use. The table is then dropped when the session ends. There is no explicit drop support. A common use for a global temporary table is to pass results into and out of procedures.

Creation syntax

```
CREATE GLOBAL TEMPORARY TABLE name (column type [NOT NULL], ... [PRIMARY KEY (column, ...)]) OPTIONS (UPDATABLE
'true')
```

- If the SERIAL data type is used, then each session's instance of the global temporary table will use it's own sequence.

See the CREATE TABLE DDL statement for all syntax options.

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

Global and Local Temporary Table Features

Primary Key Support:

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

Transaction Support:

- Temp tables support a READ_UNCOMMITED transaction isolation level. There are no locking mechanisms available to support higher isolation levels and the result of a rollback may be inconsistent across multiple transactions. If concurrent transactions are not associated with the same local temporary table or session, then the transaction isolation level is effectively SERIALIZABLE. If you want full consistency with local temporary tables, then only use a connection with 1 transaction at a time. This mode of operation is ensured by connection pooling that tracks connections by transaction.

Limitations:

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

Foreign Temporary Tables

Unlike Teiid local or global temporary tables, a foreign temporary table is a reference to a source table that is created at runtime rather than during the metadata load.

A foreign temporary table requires explicit creation syntax:

```
CREATE FOREIGN TEMPORARY TABLE name ... ON schema
```

Where the table creation body syntax is the same as a standard CREATE FOREIGN TABLE DDL statement. In general, usage of DDL OPTION clauses may be required to properly access the source table, including setting the name in source, updatability, native types, etc.

The schema name must specify an existing schema/model in the VDB. The table will be accessed as if it is on that source, however within Teiid the temporary table will still be scoped the same as a non-foreign temporary table. This means that the foreign temporary table will not belong to a Teiid schema and will be scoped to the session or procedure block where created.

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

Note	Neither a CREATE nor a corresponding DROP of a foreign temporary table issue a pushdown command, rather this mechanism simply exposes a source table for use within Teiid on a temporary basis.

There are two usage scenarios for a FOREIGN TEMPORARY TABLE. The first is to dynamically access additional tables on the source. The other is to replace the usage of a Teiid local temporary table for performance reasons. The usage pattern for the latter case would look like:

```
//- create the source table
source.native("CREATE GLOBAL TEMPORARY TABLE name IF NOT EXISTS ... ON COMMIT DELETE ROWS");
//- bring the table into Teiid
CREATE FOREIGN TEMPORARY TABLE name ... OPTIONS (UPDATABLE true)
//- use the table
...
//- forget the table
DROP TABLE name
```

Note the usage of the native procedure to pass source specific CREATE ddl to the source. Teiid does not currently attempt to pushdown a source creation of a temporary table based upon the CREATE statement. Some other mechanism, such as the native procedure shown above, must be used to first create the table. Also note the table is explicitly marked as updatable, since DDL defined tables are not updatable by default.

The source's handling of temporary tables must also be understood to make this work as intended. Sources that use the same GLOBAL table definition for all sessions while scoping the data to be session specific (such as Oracle) or sources that support session scoped temporary tables (such as PostgreSQL) will work if accessed under a transaction. A transaction is necessary because:

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

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

Alter View

Usage:

```
ALTER VIEW name AS queryExpression
```

Syntax Rules:

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

Alter Procedure

Usage:

```
ALTER PROCEDURE name AS block
```

Syntax Rules:

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

Alter Trigger

Usage:

```
ALTER TRIGGER ON name INSTEAD OF INSERT|UPDATE|DELETE (AS FOR EACH ROW block) | (ENABLED|DISABLED)
```

Syntax Rules:

- The target, name, must be an updatable view.
- Triggers are not yet true schema objects. They are scoped only to their view and have no name.
- An Update Procedures (Triggers) must already exist for the given trigger event.

Note	If the default inherent update is chosen in Teiid Designer, any SQL associated with update (shown in a greyed out text box) is not part of the VDB and cannot be enabled with an alter trigger statement.

XML SELECT Command

NOTE:XML Documents Models are deprecated. You should use SQL/XML functions or OData access to form complex documents instead.

Complex XML documents can be dynamically constructed by Teiid using XML Document Models. A document model is generally created from a schema. The document model is bound to relevant SQL statements through mapping classes. See the Designer guide for more on creating document models.

XML documents may also created via XQuery with the XMLQuery function or with various other SQL/XML functions.

Querying XML documents is similar to querying relational tables. An idiomatic SQL variant with special scalar functions gives control over which parts of a given document to return.

Query Structure

A valid XML SELECT Command against a document model is of the form SELECT … FROM … WHERE … ORDER BY … . The use of any other SELECT command clause is not allowed.

The fully qualified name for an XML element is:

```
"model"."document name".[path to element]."element name"
```

.

The fully qualified name for an attribute is:

```
"model"."document name".[path to element]."element name".[@]"attribute name"
```

Partially qualified names for elements and attributes can be used as long as the partial name is unique.

FROM Clause

Specifies the document to generate. Document names resemble other virtual groups - "model"."document name".

Syntax Rules:

- The from may only contain one unary clause specifying the desired document.

SELECT Clause

The select clause determines which parts of the XML document are generated for output.

Example Syntax:

```
select * from model.doc
```

```
select model.doc.root.parent.element.* from model.doc
```

```
select element, element1.@attribute from model.doc
```

Syntax Rules:

- SELECT * and SELECT "xml" are equivalent and specify that every element and attribute of the document should be output.
- The SELECT clause of an XML Query may only contain *, "xml", or element and attribute references from the specified document. Any other expressions are not allowed.
- If the SELECT clause contains an element or attribute reference (other than * or "xml") then only the specified elements, attributes, and their ancestor elements will be in the generated document.
- element.* specifies that the element, it’s attribute, and all child content should be output.

WHERE Clause

The where clause specifies how to filter content from the generated document based upon values contained in the underlying mapping classes. Most predicates are valid in an XML SELECT Command, however combining value references from different parts of the document may not always be allowed.

Criteria is logically applied to a context which is directly related to a mapping class. Starting with the root mapping class, there is a root context that describes all of the top level repeated elements that will be in the output document. Criteria applied to the root or any other context will change the related mapping class query to apply the affects of the criteria, which can include checking values from any of the descendant mapping classes.

Example Syntax:

```
select element, element1.@attribute from model.doc where element1.@attribute = 1
```

```
 select element, element1.@attribute from model.doc where context(element1, element1.@attribute) = 1
```

Syntax Rules:

- Each criteria conjunct must refer to a single context and can be criteria that applies to a mapping class, contain a rowlimit function, or contain rowlimitexception function.
- Criteria that applies to a mapping class is associated to that mapping class via the context function. The absence of a context function implies the criteria applies to the root context.
- At a given context the criteria can span multiple mapping classes provided that all mapping classes involved are either parents of the context, the context itself, or a descendant of the context.

Note	**Sibling Root Mapping Classes** - Implied root context user criteria against a document model with sibling root mapping classes is not generally semantically correct. It is applied as if each of the conjuncts is applied to only a single root mapping class. This behavior is the same as prior releases but may be fixed in a future release.

XML SELECT Command Specific Functions

XML SELECT Command functions are resemble scalar functions, but act as hints in the WHERE clause. These functions are only valid in an XML SELECT Command.

Context Function

```
CONTEXT(arg1, arg2)
```

Select the context for the containing conjunct.

Syntax Rules:

- Context functions apply to the whole conjunct.
- The first argument must be an element or attribute reference from the mapping class whose context the criteria conjunct will apply to.
- The second parameter is the return value for the function.

Rowlimit Function

```
ROWLIMIT(arg)
```

Limits the rows processed for the given context.

Syntax Rules:

- The first argument must be an element or attribute reference from the mapping class whose context the row limit applies.
- The rowlimit function must be used in equality comparison criteria with the right hand expression equal to an positive integer number or rows to limit.
- Only one row limit or row limit exception may apply to a given context.

Rowlimitexception Function

Limits the rows processed for the given context and throws an exception if the given number of rows is exceeded.

```
ROWLIMITEXCEPTION(arg)
```

Syntax Rules:

- The first argument must be an element or attribute reference from the mapping class whose context the row limit exception applies.
- The rowlimitexception function must be used in equality comparison criteria with the right hand expression equal to an positive integer number or rows to limit.
- Only one row limit or row limit exception may apply to a given context.

ORDER BY Clause

The XML SELECT Command ORDER BY Clause specifies ordering for the referenced mapping class queries.

Syntax Rules:

- Each order by item must be an element or attribute reference tied a output value from a mapping class.
- The order or the order by items is the relative order they will be applied to their respective mapping classes.

Document Generation

Document generation starts with the root mapping class and proceeds iteratively and hierarchically over all of the child mapping classes. This can result in a large number of query executions. For example if a document has a root mapping class with 3 child mapping classes. Then for each row selected by the root mapping class after the application of the root context criteria, each of the child mapping classes queries will also be executed.

Note	**Document Correctness** - By default XML generated by XML documents are not checked for correctness vs. the relevant schema. It is possible that the mapping class queries, the usage of specific SELECT or WHERE clause values will generated a document that is not valid with respect to the schema. See Document Validation on how to ensure correctness.

Sibling or cousin elements defined by the same mapping class that do not have a common parent in that mapping class will be treated as independent mapping classes during planning and execution. This allows for a more document centric approach to applying criteria and order bys to mapping classes.

Document Validation

The execution property XMLValidation should be set to 'true' to indicate that generated documents should be checked for correctness. Correctness checking will not prevent invalid documents from being generated, since correctness is checked after generation and not during.

Procedures

Procedure Language

Teiid supports a procedural language for defining Virtual Procedures. These are similar to stored procedures in relational database management systems. You can use this language to define the transformation logic for decomposing INSERT, UPDATE, and DELETE commands against views; these are known as Update Procedures (Triggers).

Table of Contents

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

Command Statement

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

Usage:

```
command [(WITH|WITHOUT) RETURN];
```

Example Command Statements

```
SELECT * FROM MySchema.MyTable WHERE ColA > 100 WITHOUT RETURN;
INSERT INTO MySchema.MyTable (ColA,ColB) VALUES (50, 'hi');
```

Syntax Rules:

- EXECUTE command statements may access IN/OUT, OUT, and RETURN parameters. To access the return value the statement will have the form `var = EXEC proc…`. To access OUT or IN/OUT values named parameter syntax must be used. For example, `EXEC proc(in_param⇒'1', out_param⇒var)` will assign the value of the out parameter to the variable var. It is expected that the datatype of parameter will be implicitly convertable to the datatype of the variable.
- The RETURN clause determines if the result of the command is returnable from the procedure. WITH RETURN is the default. If the command does not return a result set or the procedure does not return a result set, the RETURN clause is ignored. If WITH RETURN is specified, the result set of the command must match the expected result set of the procedure. Only the last successfully executed statement executed WITH RETURN will be returned as the procedure result set. If there are no returnable result sets and the procedure declares that a result set will be returned, then an empty result set is returned.

Dynamic SQL Command

Dynamic SQL allows for the execution of an arbitrary SQL command in a virtual procedure. Dynamic SQL is useful in situations where the exact command form is not known prior to execution.

Usage:

```
EXECUTE IMMEDIATE <sql expression> AS <variable> <type> [, <variable> <type>]* [INTO <variable>] [USING <variab
le>=<expression> [,<variable>=<expression>]*] [UPDATE <literal>]
```

Syntax Rules:

- The sql expression must be a clob/string value less than 262144 characters.
- The "AS" clause is used to define the projected symbols names and types returned by the executed SQL string. The "AS" clause symbols will be matched positionally with the symbols returned by the executed SQL string. Non-convertible types or too few columns returned by the executed SQL string will result in an error.
- The "INTO" clause will project the dynamic SQL into the specified temp table. With the "INTO" clause specified, the dynamic command will actually execute a statement that behaves like an INSERT with a QUERY EXPRESSION. If the dynamic SQL command creates a temporary table with the "INTO" clause, then the "AS" clause is required to define the table's metadata.
- The "USING" clause allows the dynamic SQL string to contain variable references that are bound at runtime to specified values. This allows for some independence of the SQL string from the surrounding procedure variable names and input names. In the dynamic command "USING" clause, each variable is specified by short name only. However in the dynamic SQL the "USING" variable must be fully qualified to "DVAR.". The "USING" clause is only for values that will be used in the dynamic SQL as legal expressions. It is not possible to use the "USING" clause to replace table names, keywords, etc. This makes using symbols equivalent in power to normal bind (?) expressions in prepared statements. The "USING" clause helps reduce the amount of string manipulation needed. If a reference is made to a USING symbol in the SQL string that is not bound to a value in the "USING" clause, an exception will occur.
- The "UPDATE" clause is used to specify the Updating Model Count. Accepted values are (0,1,*). 0 is the default value if the clause is not specified.

Example Dynamic SQL

```
...
/* Typically complex criteria would be formed based upon inputs to the procedure.
 In this simple example the criteria is references the using clause to isolate
 the SQL string from referencing a value from the procedure directly */

DECLARE string criteria = 'Customer.Accounts.Last = DVARS.LastName';

/* Now we create the desired SQL string */
DECLARE string sql_string = 'SELECT ID, First || " " || Last AS Name, Birthdate FROM Customer.Accounts WHERE '
|| criteria;

/* The execution of the SQL string will create the #temp table with the columns (ID, Name, Birthdate).
  Note that we also have the USING clause to bind a value to LastName, which is referenced in the criteria. */
EXECUTE IMMEDIATE sql_string AS ID integer, Name string, Birthdate date INTO #temp USING LastName='some name';

/* The temp table can now be used with the values from the Dynamic SQL */
loop on (SELCT ID from #temp) as myCursor
...
```

Here is an example showing a more complex approach to building criteria for the dynamic SQL string. In short, the virtual procedure AccountAccess.GetAccounts has inputs ID, LastName, and bday. If a value is specified for ID it will be the only value used in the dynamic SQL criteria. Otherwise if a value is specified for LastName the procedure will detect if the value is a search

string. If bday is specified in addition to LastName, it will be used to form compound criteria with LastName.

Example Dynamic SQL with USING clause and dynamically built criteria string

```
...
DECLARE string crit = null;

IF (AccountAccess.GetAccounts.ID IS NOT NULL)
 crit = '(Customer.Accounts.ID = DVARS.ID)';
ELSE IF (AccountAccess.GetAccounts.LastName IS NOT NULL)
BEGIN
 IF (AccountAccess.GetAccounts.LastName == '%')
   ERROR "Last name cannot be %";
 ELSE IF (LOCATE('%', AccountAccess.GetAccounts.LastName) < 0)
   crit = '(Customer.Accounts.Last = DVARS.LastName)';
 ELSE
   crit = '(Customer.Accounts.Last LIKE DVARS.LastName)';
 IF (AccountAccess.GetAccounts.bday IS NOT NULL)
   crit = '(' || crit || ' and (Customer.Accounts.Birthdate = DVARS.BirthDay))';
END
ELSE
 ERROR "ID or LastName must be specified.";

EXECUTE IMMEDIATE 'SELECT ID, First || " " || Last AS Name, Birthdate FROM Customer.Accounts WHERE ' || crit US
ING ID=AccountAccess.GetAccounts.ID, LastName=AccountAccess.GetAccounts.LastName, BirthDay=AccountAccess.GetAcc
ounts.Bday;
...
```

Known Limitations and Work-Arounds

The use of dynamic SQL command results in an assignment statement requires the use of a temp table.

Example Assignment

```
EXECUTE IMMEDIATE <expression> AS x string INTO #temp;
DECLARE string VARIABLES.RESULT = (SELECT x FROM #temp);
```

The construction of appropriate criteria will be cumbersome if parts of the criteria are not present. For example if "criteria" were already NULL, then the following example results in "criteria" remaining NULL.

Example Dangerous NULL handling

```
...
criteria = '(' || criteria || ' and (Customer.Accounts.Birthdate = DVARS.BirthDay))';
```

The preferred approach is for the user to ensure the criteria is not NULL prior its usage. If this is not possible, a good approach is to specify a default as shown in the following example.

Example NULL handling

```
...
criteria = '(' || nvl(criteria, '(1 = 1)') || ' and (Customer.Accounts.Birthdate = DVARS.BirthDay))';
```

If the dynamic SQL is an UPDATE, DELETE, or INSERT command, the rowcount of the statement can be obtained from the rowcount variable.

Example with AS and INTO clauses

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

- Unless used in other parts of the procedure, tables in the dynamic command will not be seen as sources in the Designer.

- When using the "AS" clause only the type information will be available to the Designer. ResultSet columns generated from the "AS" clause then will have a default set of properties for length, precision, etc.

Declaration Statement

A declaration statement declares a variable and its type. After you declare a variable, you can use it in that block within the procedure and any sub-blocks. A variable is initialized to null by default, but can also be assigned the value of an expression as part of the declaration statement.

Usage:

```
DECLARE <type> [VARIABLES.]<name> [= <expression>];
```

Example Syntax

```
  declare integer x;
  declare string VARIABLES.myvar = 'value';
```

Syntax Rules:

- You cannot redeclare a variable with a duplicate name in a sub-block
- The VARIABLES group is always implied even if it is not specified.
- The assignment value follows the same rules as for an Assignment Statement.
- In addition to the standard types, you may specify EXCEPTION if declaring an exception variable.

Assignment Statement

An assignment statement assigns a value to a variable by evaluating an expression.

Usage:

```
<variable reference> = <expression>;
```

Example Syntax

```
myString = 'Thank you';
VARIABLES.x = (SELECT Column1 FROM MySchema.MyTable);
```

Valid variables for assignment include any in scope variable that has been declared with a declaration statement, or the procedure in_out and out parameters. In_out and out parameters can be accessed as their fully qualified name.

Example Out Parameter

```
CREATE VIRTUAL PROCEDURE proc (OUT STRING x, INOUT STRING y) AS
BEGIN
  proc.x = 'some value ' || proc.y;
  y = 'some new value';
END
```

Special Variables

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

Usage:

Sample Usage

```
...
UPDATE FOO SET X = 1 WHERE Y = 2;
DECLARE INTEGER UPDATED = VARIABLES.ROWCOUNT;
...
```

Non-update command statements (WITH or WITHOUT RETURN) will reset the ROWCOUNT to 0.

Note	To ensure you are getting the appropriate ROWCOUNT value, save the ROWCOUNT to a variable immediately after the command statement.

Compound Statement

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

Usage:

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

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

Syntax Rules

- IF NOT ATOMIC or no ATOMIC clause is specified, the block will be executed non-atomically.
- IF ATOMIC the block must execute atomically. If a transaction is already associated with the thread, no additional action will be taken - savepoints and/or sub-transactions are not currently used. If the higher level transaction is used and the block does not complete - regardless of the presence of exception handling the transaction will be marked as rollback only. Otherwise a transaction will be associated with the execution of the block. Upon successful completion of the block the transaction will be committed.
- The label must not be the same as any other label used in statements containing this one.
- Variable assignments and the implicit result cursor are unaffected by rollbacks. If a block does not complete successfully its assignments will still take affect.

Exception Handling

If the EXCEPTION clause is used with in a compound statement, any processing exception emitted from statements will be caught with the flow of execution transferring to EXCEPTION statements. Any block level transaction started by this block will commit if the exception handler successfully completes. If another exception or the original exception is emitted from the exception handler the transaction will rollback. Any temporary tables or variables specific to the BLOCK will not be available to the exception handler statements.

Note	Only processing exceptions, which are typically caused by errors originating at the sources or with function execution, are caught. A low-level internal Teiid error or Java `RuntimeException` will not be caught.

To aid in the processing of a caught exception the EXCEPTION clause specifies a group name that exposes the significant fields of the exception. The exception group will contain:

Variable	Type	Description
STATE	string	The SQL State
ERRORCODE	integer	The error or vendor code. In the case of Teiid internal exceptions this will be the integer suffix of the TEIIDxxxx code
TEIIDCODE	string	The full Teiid event code. Typically TEIIDxxxx.
EXCEPTION	object	The exception being caught, will be an instance of `TeiidSQLException`
CHAIN	object	The chained exception or cause of the current exception

Note	Teiid does not yet fully comply with the ANSI SQL specification on SQL State usage. For Teiid errors without an underlying SQLException cause, it is best to use the Teiid code.

The exception group name may not be the same as any higher level exception group or loop cursor name.

Example Exception Group Handling

```
BEGIN
    DECLARE EXCEPTION e = SQLEXCEPTION 'this is bad' SQLSTATE 'xxxxx';
    RAISE variables.e;
EXCEPTION e
    IF (e.state = 'xxxxx')
        //in this trivial example, we'll always hit this branch and just log the exception
        RAISE SQLWARNING e.exception;
    ELSE
        RAISE e.exception;
END
```

If Statement

An IF statement evaluates a condition and executes either one of two statements depending on the result. You can nest IF statements to create complex branching logic. A dependent ELSE statement will execute its statement only if the IF statement evaluates to false.

Usage:

```
IF (criteria)
   block
[ELSE
   block]
END
```

Example If Statement

```
IF ( var1 = 'North America')
BEGIN
  ...statement...
END ELSE
BEGIN
  ...statement...
END
```

The criteria may be any valid boolean expression or an IS DISTINCT FROM predicate. IS DISTINCT FROM uses the syntax:

```
rowVal IS [NOT] DISTINCT FROM rowValOther
```

Where rowVal and rowValOther are references to row value group. This would typically be used in instead of update triggers on views to quickly determine if the row values are changing:

Example IS DISTINCT FROM If Statement

```
IF ( "new" IS DISTINCT FROM "old")
BEGIN
  ...statement...
END
```

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

Tip	NULL values should be considered in the criteria of an IF statement. IS NULL criteria can be used to detect the presence of a NULL value.

Loop Statement

A LOOP statement is an iterative control construct that is used to cursor through a result set.

Usage:

```
[label :] LOOP ON <select statement> AS <cursorname>
    statement
```

Syntax Rules

- The label must not be the same as any other label used in statements containing this one.

While Statement

A WHILE statement is an iterative control construct that is used to execute a statement repeatedly whenever a specified condition is met.

Usage:

```
[label :] WHILE <criteria>
    statement
```

Syntax Rules

- The label must not be the same as any other label used in statements containing this one.

Continue Statement

A CONTINUE statement is used inside a LOOP or WHILE construct to continue with the next loop by skipping over the rest of the statements in the loop. It must be used inside a LOOP or WHILE statement.

Usage:

```
CONTINUE [label];
```

Syntax Rules

- If the label is specified, it must exist on a containing LOOP or WHILE statement.
- If no label is specified, the statement will affect the closest containing LOOP or WHILE statement.

Break Statement

A BREAK statement is used inside a LOOP or WHILE construct to break from the loop. It must be used inside a LOOP or WHILE statement.

Usage:

```
BREAK [label];
```

Syntax Rules

- If the label is specified, it must exist on a containing LOOP or WHILE statement.
- If no label is specified, the statement will affect the closest containing LOOP or WHILE statement.

Leave Statement

A LEAVE statement is used inside a compound, LOOP, or WHILE construct to leave to the specified level.

Usage:

```
LEAVE label;
```

Syntax Rules

- The label must exist on a containing compound statement, LOOP, or WHILE statement.

Return Statement

A Return statement gracefully exits the procedure and optionally returns a value.

Usage:

```
RETURN [expression];
```

Syntax Rules

- If an expression is specified, the procedure must have a return parameter and the value must be implicitly convertable to the expected type.
- Even if the procedure has a return value, it is not required to specify a return value in a RETURN statement.

Error Statement

An ERROR statement declares that the procedure has entered an error state and should abort. This statement will also roll back the current transaction, if one exists. Any valid expression can be specified after the ERROR keyword.

Usage:

```
ERROR message;
```

Example Error Statement

```
ERROR 'Invalid input value: ' || nvl(Acct.GetBalance.AcctID, 'null');
```

An ERROR statement is equivalent to:

```
RAISE SQLEXCEPTION message;
```

Raise Statement

A RAISE statement is used to raise an exception or warning. When raising an exception, this statement will also roll back the current transaction, if one exists.

Usage:

```
RAISE [SQLWARNING] exception;
```

Where exception may be a variable reference to an exception or an exception expression.

Syntax Rules

- If SQLWARNING is specified, the exception will be sent to the client as a warning and the procedure will continue to execute.
- A null warning will be ignored. A null non-warning exception will still cause an exception to be raised.

Example Raise Statement

```
RAISE SQLWARNING SQLEXCEPTION 'invalid' SQLSTATE '05000';
```

Exception Expression

An exception expression creates an exception that can be raised or used as a warning.

Usage:

```
SQLEXCEPTION message [SQLSTATE state [, code]] CHAIN exception
```

Syntax Rules

- Any of the values may be null;
- message and state are string expressions specifying the exception message and SQL state respectively. Teiid does not yet fully comply with the ANSI SQL specification on SQL state usage, but you are allowed to set any SQL state you choose.
- code is an integer expression specifying the vendor code
- exception must be a variable reference to an exception or an exception expression and will be chained to the resulting exception as its parent.

Virtual Procedures

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

Table of Contents

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

Virtual Procedure Definition

In Designer:

```
[CREATE VIRTUAL PROCEDURE] statement
```

In DDL: DDL Metadata#Create Procedure/Function

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

There is no explicit cursoring or value returning statement, rather the last unnamed command statement executed in the procedure that returns a result set will be returned as the result. The output of that statement must match the expected result set and parameters of the procedure.

Procedure Parameters

Virtual procedures can take zero or more IN/INOUT parameters and may also have any number of OUT parameters and an optional RETURN parameter. Each input has the following information that is used during runtime processing:

- Name - The name of the input parameter
- Datatype - The design-time type of the input parameter
- Default value - The default value if the input parameter is not specified
- Nullable - NO_NULLS, NULLABLE, NULLABLE_UNKNOWN; parameter is optional if nullable, and is not required to be listed when using named parameter syntax

You reference a parameter in a virtual procedure by using the fully-qualified name of the param (or less if unambiguous). For example, MySchema.MyProc.Param1.

Example of Referencing an Input Parameter and Assigning an Out Parameter for `GetBalance` Procedure

```
BEGIN
  MySchema.GetBalance.RetVal = UPPER(MySchema.GetBalance.AcctID);
  SELECT Balance FROM MySchema.Accts WHERE MySchema.Accts.AccountID = MySchema.GetBalance.AcctID;
END
```

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

Example Virtual Procedures

This example is a LOOP that walks through a cursored table and uses CONTINUE and BREAK.

Virtual Procedure Using LOOP, CONTINUE, BREAK

```
BEGIN
  DECLARE double total;
  DECLARE integer transactions;
  LOOP ON (SELECT amt, type FROM CashTxnTable) AS txncursor
  BEGIN
    IF(txncursor.type <> 'Sale')
    BEGIN
      CONTINUE;
    END ELSE
    BEGIN
      total = (total + txncursor.amt);
      transactions = (transactions + 1);
      IF(transactions = 100)
      BEGIN
        BREAK;
      END
    END
  END
  SELECT total, (total / transactions) AS avg_transaction;
END
```

This example is uses conditional logic to determine which of two SELECT statements to execute.

Virtual Procedure with Conditional SELECT

```
BEGIN
  DECLARE string VARIABLES.SORTDIRECTION;
  VARIABLES.SORTDIRECTION = PartsVirtual.OrderedQtyProc.SORTMODE;
  IF ( ucase(VARIABLES.SORTDIRECTION) = 'ASC' )
  BEGIN
    SELECT * FROM PartsVirtual.SupplierInfo WHERE QUANTITY > PartsVirtual.OrderedQtyProc.QTYIN ORDER BY PartsVi
rtual.SupplierInfo.PART_ID;
  END ELSE
  BEGIN
    SELECT * FROM PartsVirtual.SupplierInfo WHERE QUANTITY > PartsVirtual.OrderedQtyProc.QTYIN ORDER BY PartsVi
rtual.SupplierInfo.PART_ID DESC;
  END
END
```

Executing Virtual Procedures

You execute procedures using the SQL EXECUTE command. If the procedure has defined inputs, you specify those in a sequential list, or using "name=value" syntax. You must use the name of the input parameter, scoped by the full procedure name if the parameter name is ambiguous in the context of other columns or variables in the procedure.

A virtual procedure call will return a result set just like any SELECT, so you can use this in many places you can use a SELECT. Typically you'll use the following syntax:

```
SELECT * FROM (EXEC ...) AS x
```

Limitations

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

Triggers

View Triggers

Views are abstractions above physical sources. They typically union or join information from multiple tables, often from multiple data sources or other views. Teiid can perform update operations against views. Update commands - INSERT, UPDATE, or DELETE - against a view require logic to define how the tables and views integrated by the view are affected by each type of command. This transformation logic, also referred to as a **trigger**, is invoked when an update command is issued against a view. Update procedures define the logic for how a user's update command against a view should be decomposed into the individual commands to be executed against the underlying physical sources. Similar to Virtual Procedures, update procedures have the ability to execute queries or other commands, define temporary tables, add data to temporary tables, walk through result sets, use loops, and use conditional logic.

Teiid supports INSTEAD OF triggers on views similar to traditional databases. There may only be 1 FOR EACH ROW procedure for each INSERT, UPDATE, or DELETE operation against a view.

Usage:

```
CREATE TRIGGER ON view_name INSTEAD OF INSERT|UPDATE|DELETE AS
FOR EACH ROW
...
```

Update Procedure Processing

1. The user application submits the SQL command through one of SOAP, JDBC, or ODBC.
2. The view this SQL command is executed against is detected.
3. The correct procedure is chosen depending upon whether the command is an INSERT, UPDATE, or DELETE.
4. The procedure is executed. The procedure itself can contain SQL commands of its own which can be of different types than the command submitted by the user application that invoked the procedure.
5. Commands, as described in the procedure, are issued to the individual physical data sources or other views.
6. A value representing the number of rows changed is returned to the calling application.

Source Triggers

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

Usage:

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

For Each Row

Only the FOR EACH ROW construct is supported as a trigger handler. A FOR EACH ROW trigger procedure will evaluate its block for each row of the view/source affected by the associated change. For UPDATE and DELETE statements this will be every row that passes the WHERE condition. For INSERT statements there will be 1 new row for each set of values from the VALUES or query expression. For a view the rows updated is reported as this number regardless of the affect of the underlying procedure logic.

Definition

Usage:

```
FOR EACH ROW
   BEGIN ATOMIC
      ...
   END
```

The BEGIN and END keywords are used to denote block boundaries. Within the body of the procedure, any valid statement may be used.

Tip	The use of the atomic keyword is currently optional for backward compatibility, but unlike a normal block, the default for instead of triggers is atomic.

Special Variables

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

NEW Variables

Every attribute on the view/table whose UPDATE and INSERT transformations you are defining has an equivalent variable named NEW.<column_name>

When an INSERT or an UPDATE command is executed or the event is received, these variables are initialized to the values in the INSERT VALUES clause or the UPDATE SET clause respectively.

In an UPDATE procedure, the default value of these variables, if they are not set by the command, is the old value. In an INSERT procedure, the default value of these variables is the default value of the virtual table attributes. See CHANGING Variables for distinguishing defaults from passed values.

OLD Variables

Every attribute on the view/table whose UPDATE and DELETE transformations you are defining has an equivalent variable named OLD.<column_name>

When a DELETE or UPDATE command is executed or the event is received, these variables are initialized to the current values of the row being deleted or updated respectively.

CHANGING Variables

Every attribute on the view/table whose UPDATE and INSERT transformations you are defining has an equivalent variable named CHANGING.<column_name>

When an INSERT or an UPDATE command is executed or an the event is received, these variables are initialized to `true` or `false` depending on whether the INPUT variable was set by the command. A CHANGING variable is commonly used to differentiate between a default insert value and one specified in the user query.

For example, for a view with columns A, B, C:

If User Executes…	Then…
`INSERT INTO VT (A, B) VALUES (0, 1)`	CHANGING.A = true, CHANGING.B = true, CHANGING.C = false
`UPDATE VT SET C = 2`	CHANGING.A = false, CHANGING.B = false, CHANGING.C = true

Comments

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

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

And single line comments:

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

Comment nesting is supported.

Datatypes

Supported Types

Teiid supports a core set of runtime types. Runtime types can be different than semantic types defined in type fields at design time. The runtime type can also be specified at design time or it will be automatically chosen as the closest base type to the semantic type.

Table 1. **Teiid Runtime Types**

Type	Description	Java Runtime Class	JDBC Type	ODBC Type
string or varchar	variable length character string with a maximum length of 4000.	java.lang.String	VARCHAR	VARCHAR
varbinary	variable length binary string with a nominal maximum length of 8192.	byte[] [1]	VARBINARY	VARBINARY
char	a single Unicode character	java.lang.Character	CHAR	CHAR
boolean	a single bit, or Boolean, that can be true, false, or null (unknown)	java.lang.Boolean	BIT	SMALLINT
byte or tinyint	numeric, integral type, signed 8-bit	java.lang.Byte	TINYINT	SMALLINT
short or smallint	numeric, integral type, signed 16-bit	java.lang.Short	SMALLINT	SMALLINT
integer or serial	numeric, integral type, signed 32-bit. The serial type also implies not null and has an auto-incrementing value that starts at 1. serial types are not automatically UNIQUE.	java.lang.Integer	INTEGER	INTEGER
long or bigint	numeric, integral type, signed 64-bit	java.lang.Long	BIGINT	NUMERIC
biginteger	numeric, integral type, arbitrary precision of up to 1000 digits	java.math.BigInteger	NUMERIC	NUMERIC
float or real	numeric, floating point type, 32-bit IEEE 754 floating-point numbers	java.lang.Float	REAL	FLOAT

double	numeric, floating point type, 64-bit IEEE 754 floating-point numbers	java.lang.Double	DOUBLE	DOUBLE
bigdecimal or decimal	numeric, floating point type, arbitrary precision of up to 1000 digits.	java.math.BigDecimal	NUMERIC	NUMERIC
date	datetime, representing a single day (year, month, day)	java.sql.Date	DATE	DATE
time	datetime, representing a single time (hours, minutes, seconds, milliseconds)	java.sql.Time	TIME	TIME
timestamp	datetime, representing a single date and time (year, month, day, hours, minutes, seconds, milliseconds, nanoseconds)	java.sql.Timestamp	TIMESTAMP	TIMESTAMP
object	any arbitrary Java object, must implement java.lang.Serializable	Any	JAVA_OBJECT	VARCHAR
blob	binary large object, representing a stream of bytes	java.sql.Blob [2]	BLOB	VARCHAR
clob	character large object, representing a stream of characters	java.sql.Clob [3]	CLOB	VARCHAR
xml	XML document	java.sql.SQLXML[4]	JAVA_OBJECT	VARCHAR
geometry	Geospatial Object	java.sql.Blob [5]	BLOB	BLOB

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.

Reference Link

1. The runtime type is org.teiid.core.types.BinaryType. Translators will need to explicitly handle BinaryType values. UDFs will instead have a byte[] value passed.
2. The concrete type is expected to be org.teiid.core.types.BlobType
3. The concrete type is expected to be org.teiid.core.types.ClobType
4. The concrete type is expected to be org.teiid.core.types.XMLType
5. The concrete type is expected to be org.teiid.core.types.GeometryType

Arrays

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

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

Example array types:

```
string[]
```

```
integer[][]
```

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

Type Conversions

Data types may be converted from one form to another either explicitly or implicitly. Implicit conversions automatically occur in criteria and expressions to ease development. Explicit datatype conversions require the use of the **CONVERT** function or **CAST** keyword.

Type Conversion Considerations:

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

For example:

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

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

- Array conversions are currently only valid to and from compatible object arrays. It is not yet possible for example to cast from integer[] to long[].
- Explicit conversions that are not allowed between two types will result in an exception before execution. Allowed explicit conversions may still fail during processing if the runtime values are not actually convertable.

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

Table 1. **Type Conversions**

Source Type	Valid Implicit Target Types	Valid Explicit Target Types
string	clob	char, boolean, byte, short, integer, long, biginteger, float, double, bigdecimal, xml [1]
char	string	
boolean	string, byte, short, integer, long, biginteger, float, double, bigdecimal	
byte	string, short, integer, long, biginteger, float, double, bigdecimal	boolean
short	string, integer, long, biginteger, float,	boolean, byte

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
xml		string [3]

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

Special Conversion Cases

Conversion of String Literals

Teiid automatically converts string literals within a SQL statement to their implied types. This typically occurs in a criteria comparison where an expression with a different datatype is compared to a literal string:

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

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

Converting to Boolean

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

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

Date/Time/Timestamp Type Conversions

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

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

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

Escaped Literal Syntax

Rather than relying on implicit conversion, datatype values may be expressed directly in SQL using escape syntax to define the type. Note that the supplied string value must match the expected format exactly or an exception will occur.

Datatype	Escaped Syntax
DATE	\{d 'yyyy-mm-dd'}
TIME	\{t 'hh-mm-ss'}
TIMESTAMP	\{ts 'yyyy-mm-dd[hh:mm:ss.[fff…]]'}

Updatable Views

Any view may be marked as updatable. In many circumstances the view definition may allow the view to be inherently updatable without the need to manually define a trigger to handle INSERT/UPDATE/DELETE operations.

An inherently updatable view cannot be defined with a query that has:

- A set operation (INTERSECT, EXCEPT, UNION).
- SELECT DISTINCT
- Aggregation (aggregate functions, GROUP BY, HAVING)
- A LIMIT clause

A UNION ALL can define an inherently updatable view only if each of the UNION branches is itself inherently updatable. A view defined by a UNION ALL can support inherent INSERTs if it is a Federated Optimizations#Partitioned Union and the INSERT specifies values that belong to a single partition.

Any view column that is not mapped directly to a column is not updatable and cannot be targeted by an UPDATE set clause or be an INSERT column.

If a view is defined by a join query or has a WITH clause it may still be inherently updatable. However in these situations there are further restrictions and the resulting query plan may execute multiple statements. For a non-simple query to be updatable, it is required:

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

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

preserved Table

A key-preserved table has a primary or unique key that would remain unique if it were projected into the result of the query. Note that it is not actually required for a view to reference the key columns in the SELECT clause. The query engine can detect a key preserved table by analyzing the join structure. The engine will ensure that a join of a key-preserved table must be against one of its foreign keys.

Transaction Support

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

Table 1. **Teiid Transaction Scopes**

Scope	Description
Command	Treats the user command as if all source commands are executed within the scope of the same transaction. The AutoCommitTxn execution property controls the behavior of command level transactions.
Local	The transaction boundary is local defined by a single client session.
Global	Teiid participates in a global transaction as an XA Resource.

The default transaction isolation level for Teiid is READ_COMMITTED.

AutoCommitTxn Execution Property

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

Table 1. **AutoCommitTxn Settings**

Setting	Description
OFF	Do not wrap each command in a transaction. Individual source commands may commit or rollback regardless of the success or failure of the overall command.
ON	Wrap each command in a transaction. This mode is the safest, but may introduce performance overhead.
DETECT	This is the default setting. Will automatically wrap commands in a transaction, but only if the command seems to be transactionally unsafe.

The concept of command safety with respect to a transaction is determined by Teiid based upon command type, the transaction isolation level, and available metadata. A wrapping transaction is not needed if:

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

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

Updating Model Count

The term "updating model count" refers to the number of times any model is updated during the execution of a command. It is used to determine whether a transaction, of any scope, is required to safely execute the command.

Table 1. **Updating Model Count Settings**

Count	Description
0	No updates are performed by this command.
1	Indicates that only one model is updated by this command (and its subcommands). Also the success or failure of that update corresponds to the success of failure of the command. It should not be possible for the update to succeed while the command fails. Execution is not considered transactionally unsafe.
*	Any number greater than 1 indicates that execution is transactionally unsafe and an XA transaction will be required.

JDBC and Transactions

JDBC API Functionality

The transaction scopes above map to these JDBC modes:

- **Command** - Connection autoCommit property set to true.
- **Local** - Connection autoCommit property set to false. The transaction is committed by setting autoCommit to true or calling java.sql.Connection.commit . The transaction can be rolled back by a call to java.sql.Connection.rollback
- **Global** - the XAResource interface provided by an XAConnection is used to control the transaction. Note that XAConnections are available only if Teiid is consumed through its XADataSource, `org.teiid.jdbc.TeiidDataSource` . JEE containers or data access APIs typically control XA transactions on behalf of application code.

J2EE Usage Models

J2EE provides three ways to manage transactions for beans:

- **Client-controlled** – the client of a bean begins and ends a transaction explicitly.
- **Bean-managed** – the bean itself begins and ends a transaction explicitly.
- **Container-managed** – the app server container begins and ends a transaction automatically.

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

Transactional Behavior with WildFly Data Source Types

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

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

Note	Teiid Designer creates "local" data source by default which is not optimal for the XA transactions. Teiid would like this to be creating a XA data sources, however with current limitations with DTP that feature is currently not available. To create XA data source, look in the WildFly "doc" directory for example templates, or use the "admin-console" to create the XA data sources.

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

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

Table 1. **Teiid Transaction Participation**

Teiid-Tx-Scope	XA source	Local Source	No-Tx Source
Local (Auto-commit=false)	always	Only If Single Source	never
Global	always	Only If Single Source	never
Auto-commit=true, AutoCommitTxn=ON, or DETECT and txn started	always	Only If Single Source	never
Auto-commit=true, AutoCommitTxn=OFF	never	never	never

Limitations and Workarounds

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

Data Roles

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

Prior to applying data roles, you should consider restricting source system access through the fundamental design of your VDB. Foremost, Teiid can only access source entries that are represented in imported metadata. You should narrow imported metadata to only what is necessary for use by your VDB. When using Teiid Designer, you may then go further and modify the imported metadata at a granular level to remove specific columns, mark tables as non-updatable, etc.

If data role validation is enabled and data roles are defined in a VDB, then access permissions will be enforced by the Teiid Server. The use of data roles may be disabled system wide by removing the setting for the teiid subsystem policy-decider-module. Data roles also have built-in system functions that can be used for row-based and other authorization checks.

Tip	Unlike previous versions of Teiid data roles will only be checked if present in a VDB. A VDB deployed without data roles is open for use by any authenticated user. If you want to ensure some attempt has been made at securing access, then set the data-roles-required configuration element to true via the CLI or in the standalone.xml on the teiid subsystem.

Permissions

User Query Permissions

CREATE, READ, UPDATE, DELETE (CRUD) permissions can be set for any resource path in a VDB. A resource path can be as specific as the fully qualified name of a column or as general a top level model (schema) name. Permissions granted to a particular path apply to it and any resource paths that share the same partial name. For example, granting read to "model" will also grant read to "model.table", "model.table.column", etc. Allowing or denying a particular action is determined by searching for permissions from the most to least specific resource paths. The first permission found with a specific allow or deny will be used. Thus it is possible to set very general permissions at high-level resource path names and to override only as necessary at more specific resource paths.

Permission grants are only needed for resources that a role needs access to. Permissions are also only applied to the columns/tables/procedures in the user query - not to every resource accessed transitively through view and procedure definitions. It is important therefore to ensure that permission grants are applied consistently across models that access the same resources.

Note	Unlike previous versions of Teiid, non-visible models are accessible by user queries. To restrict user access at a model level, at least one data role should be created to enable data role checking. In turn that role can be mapped to any authenticated user and should not grant permissions to models that should be inaccessible.

Permissions are not applicable to the SYS and pg_catalog schemas. These metadata reporting schemas are always accessible regardless of the user. The SYSADMIN schema however may need permissions as applicable.

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

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

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

- *CREATE*- on the Table being inserted into.
- *CREATE*- on every column being inserted on that Table.

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

- *UPDATE*- on the Table being updated.
- *UPDATE*- on every column being updated on that Table.
- *READ*- on every column referenced in the criteria.

To process a *DELETE* statement, the user account requires the following access rights:

- *DELETE*- on the Table being deleted.
- *READ*- on every column referenced in the criteria.

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

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

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

- *EXECUTE (or READ)*- on the Function being called.

To process any ALTER or CREATE TRIGGER statement, the user account requires the following access rights:

- *ALTER*- on the view or procedure that is effected. INSTEAD OF Triggers (update procedures) are not yet treated as full schema objects and are instead treated as attributes of the view.

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

- *LANGUAGE* - specifying the language name that is allowed.

To process any statement against a Teiid temporary table requires the following access rights:

- allow-create-temporary-tables attribute on any applicable role
- *CREATE,READ,UPDATE,DELETE* - against the target model/schema as needed for operations against a FOREIGN temporary table.

Row and Column Based Security

Although specified in a similar way to user query CRUD permissions, row-based and column-based permissions may be used together or separately to control at a more granular and consistent level the data returned to users. See also XML Definition for examples of specifying data roles with row and column based security.

Row-Based Security

A permission against a fully qualified table/view/procedure may also specify a condition. Unlike the allow CRUD actions defined above, a condition is always applied - not just at the user query level. The condition can be any valid SQL referencing the columns of the table/view/procedure. Procedure result set columns may be referenced as proc.col. The condition will act as a row-based filter and as a checked constraint for insert/update operations.

How Row-Based Conditions Are Applied

A condition is applied conjunctively to update/delete/select where clauses against the affected resource. Those queries will therefore only ever be effective against the subset of rows that pass the condition, i.e. "SELECT * FROM TBL WHERE blah **AND condition**". The condition will be present regardless of how the table/view is used in the query, whether via a union, join, etc.

Inserts and updates against physical tables affected by a condition are further validated so that the insert/change values must pass the condition (evaluate to true) for the insert/update to succeed - this is effectively the same a SQL constraint. This will happen for all styles of insert/update - insert with query expression, bulk insert/update, etc. Inserts/updates against views are not checked with regards to the constraint. You may disable the insert/update constraint check by setting the condition constraint flag to false. This is typically only needed in circumstances when the condition cannot always be evaluated. However disabling the condition as a constraint simply drops the condition from consideration when logically evaluating the constraint. Any other condition constraints will still be evaluated.

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

Considerations When Using Conditions

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

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

In addition to managing permission conditions on a per-role basis, another approach is to add condition permissions would in an any authenticated role such that the conditions are generalized for all users/roles using the `hasRole`, `user`, and other such security functions. The advantage of the latter approach is that there is effectively a static row-based policy in effect such that all query plans can still be shared between users.

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

Limitations

- Conditions on source tables that act as check constraints must currently not contain correlated subqueries.
- Conditions may not contain aggregate or windowed functions.
- Tables and procedures referenced via subqueries will still have row-based filters and column masking applied to them.

Note	Row-based filter conditions are enforced even for materialized view loads.

You should ensure that tables consumed to produce materialized views do not have row-based filter conditions on them that could affect the materialized view results.

Column Masking

A permission against a fully qualified table/view/procedure column may also specify a mask and optionally a condition. When the query is submitted the roles are consulted and the relevant mask/condition information are combined to form a searched case expression to mask the values that would have been returned by the access. Unlike the CRUD allow actions defined above, the resulting masking effect is always applied - not just at the user query level. The condition and expression can be any valid SQL referencing the columns of the table/view/procedure. Procedure result set columns may be referenced as proc.col.

How Column Masks Are Applied

Column masking is applied only against SELECTs. Column masking is applied logically after the affect of row based security. However since both views and source tables may have row and column based security, the actual view level masking may take place on top of source level masking. If the condition is specified along with the mask, then the effective mask expression effects only a subset of the rows: "CASE WHEN condition THEN mask ELSE column". Otherwise the condition is assumed to be TRUE, meaning that the mask applies to all rows.

If multiple roles specify a mask against a column, the mask order argument will determine their precedence from highest to lowest as part of a larger searched case expression. For example a mask with the default order of 0 and a mask with an order of 1 would be combined as "CASE WHEN condition1 THEN mask1 WHEN condition0 THEN mask0 ELSE column".

Considerations When Using Masking

Non-pushdown masking conditions/expressions may adversely impact performance, since their evaluation may inhibit pushdown of query constructs on top of the affected resource. In some circumstances the insertion of masking may require that the plan be altered with the addition of an inline view, which can result in adverse performance against sources that do not support inline views.

In addition to managing masking on a per-role basis with the use of the order value, another approach is to specify masking in a single any authenticated role such that the conditions/expressions are generalized for all users/roles using the `hasRole`, `user`, and other such security functions. The advantage of the latter approach is that there is effectively a static masking policy in effect such that all query plans can still be shared between users.

Limitations

- In the event that two masks have the same order value, it is not well defined what order they are applied in.
- Masks or their conditions may not contain aggregate or windowed functions.
- Tables and procedures referenced via subqueries will still have row-based filters and column masking applied to them.

Note	Masking is enforced even for materialized view loads.

You should ensure that tables consumed to produce materialized views do not have masking on them that could affect the materialized view results.

Role Mapping

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

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

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

XML Definition

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

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

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

vdb.xml defining RoleA, RoleB, and Admin

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

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

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

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

        <permission>
            <resource-name>modelName.TableA.colum2</resource-name>
            <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.

Additional Role Attributes

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

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

Temp Table Role for Any Authenticated

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

     <permission>
          ...
     </permission>

</data-role>
```

Language Access

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

vdb.xml allowing JavaScript access

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

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

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

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

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

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

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

    </data-role>
```

```
</vdb>
```

Row-Based Security

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

vdb.xml allowing conditional access

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

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

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

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

    </data-role>

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

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

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

    </data-role>

</vdb>
```

Column Masking

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

vdb.xml with column masking

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

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

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

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

```
        </permission>

    </data-role>

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

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

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

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

    </data-role>

</vdb>
```

Customizing

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

System Schema

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

By default a system schema for ODBC metadata pg_catalog is also exposed - however that should be considered for general use.

Metadata Visibility

The SYS system schema tables and procedures are always visible/accessible.

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

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

SYS Schema

System schema for public information and actions.

Table of Contents

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

Tables/Views

SYS.Columns

This table supplies information about all the elements (columns, tags, attributes, etc) in the virtual database.

Column Name	Type	Description
VDBName	string	VDB name
SchemaName	string	Schema Name
TableName	string	Table name
Name	string	Element name (not qualified)
Position	integer	Position in group (1-based)
NameInSource	string	Name of element in source
DataType	string	Teiid runtime data type name
Scale	integer	Number of digits after the decimal point

ElementLength	integer	Element length (mostly used for strings)
sLengthFixed	boolean	Whether the length is fixed or variable
SupportsSelect	boolean	Element can be used in SELECT
SupportsUpdates	boolean	Values can be inserted or updated in the element
IsCaseSensitive	boolean	Element is case-sensitive
IsSigned	boolean	Element is signed numeric value
IsCurrency	boolean	Element represents monetary value
IsAutoIncremented	boolean	Element is auto-incremented in the source
NullType	string	Nullability: "Nullable", "No Nulls", "Unknown"
MinRange	string	Minimum value
MaxRange	string	Maximum value
DistinctCount	integer	Distinct value count, -1 can indicate unknown
NullCount	integer	Null value count, -1 can indicate unknown
SearchType	string	Searchability: "Searchable", "All Except Like", "Like Only", Unsearchable"
Format	string	Format of string value
DefaultValue	string	Default value
JavaClass	string	Java class that will be returned
Precision	integer	Number of digits in numeric value
CharOctetLength	integer	Measure of return value size
Radix	integer	Radix for numeric values
GroupUpperName	string	Upper-case full group name
UpperName	string	Upper-case element name
UID	string	Element unique ID

Description	string	Description
TableUID	string	Parent Table unique ID

SYS.DataTypes

This table supplies information on datatypes.

Column Name	Type	Description
Name	string	Teiid design-time type name
IsStandard	boolean	Always false
IsPhysical	boolean	Always false
TypeName	string	Design-time type name (same as Name)
JavaClass	string	Java class returned for this type
Scale	integer	Max scale of this type
TypeLength	integer	Max length of this type
NullType	string	Nullability: "Nullable", "No Nulls", "Unknown"
IsSigned	boolean	Is signed numeric?
IsAutoIncremented	boolean	Is auto-incremented?
IsCaseSensitive	boolean	Is case-sensitive?
Precision	integer	Max precision of this type
Radix	integer	Radix of this type
SearchType	string	Searchability: "Searchable", "All Except Like", "Like Only", "Unsearchable"
UID	string	Data type unique ID
RuntimeType	string	Teiid runtime data type name
BaseType	string	Base type
Description	string	Description of type

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)
UID	string	Key unique ID

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

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

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	

NameInSource	string	
UID	string	Function UID
Description	string	Description
IsVarArgs	boolean	Does the function accept variable arguments

SYS.Properties

This table supplies user-defined properties on all objects based on metamodel extensions. Normally, this table is empty if no metamodel extensions are being used.

Column Name	Type	Description
Name	string	Extension property name
Value	string	Extension property value
UID	string	Key unique ID
ClobValue	clob	Clob Value

SYS.ReferenceKeyColumns

This table supplies informaton about column’s key reference.

Column Name	Type	Description
PKTABLE_CAT	string	VDB Name
PKTABLE_SCHEM	string	Schema Name
PKTABLE_NAME	string	Table/View Name
PKCOLUMN_NAME	string	Column Name
FKTABLE_CAT	string	VDB Name
FKTABLE_SCHEM	string	Schema Name
FKTABLE_NAME	string	Table/View Name
FKCOLUMN_NAME	string	Column Name
KEY_SEQ	short	Key Sequence
UPDATE_RULE	integer	Update Rule
DELETE_RULE	integer	Delete Rule

FK_NAME	string	FK Name
PK_NAME	string	PK Nmae
DEFERRABILITY	integer	

SYS.Schemas

This table supplies information about all the schemas in the virtual database, including the system schema itself (System).

Column Name	Type	Description
VDBName	string	VDB name
Name	string	Schema name
IsPhysical	boolean	True if this represents a source
UID	string	Unique ID
Description	string	Description
PrimaryMetamodelURI	string	URI for the primary metamodel describing the model used for this schema

SYS.Tables

This table supplies information about all the groups (tables, views, documents, etc) in the virtual database.

Column Name	Type	Description
VDBName	string	VDB name
SchemaName	string	Schema Name
Name	string	Short group name
Type	string	Table type (Table, View, Document, …)
NameInSource	string	Name of this group in the source
IsPhysical	boolean	True if this is a source table
SupportsUpdates	boolean	True if group can be updated
UID	string	Group unique ID
Cardinality	integer	Approximate number of rows in the group
Description	string	Description

IsSystem	boolean	True if in system table
SchemaUID	string	Parent Schema unique ID

SYS.VirtualDatabases

This table supplies information about the currently connected virtual database, of which there is always exactly one (in the context of a connection).

Column Name	Type	Description
Name	string	The name of the VDB
Version	string	The version of the VDB

SYS.spatial_sys_ref

See also the PostGIS Documentation

Column Name	Type	Description
srid	integer	Spatial Reference Identifier
auth_name	string	Name of the standard or standards body
auth_srid	integer	SRID for the auth_name authority
srtext	string	Well-Known Text representation
proj4text	string	For use with the Proj4 library

SYS.GEOMETRY_COLUMNS

See also the PostGIS Documentation

Column Name	Type	Description
F_TABLE_CATALOG	string	catalog name
F_TABLE_SCHEMA	string	schema name
F_TABLE_NAME	string	table name
F_GEOMETRY_COLUMN	string	column name
COORD_DIMENSION	integer	Number of coordinate dimensions
SRID	integer	Spatial Reference Identifier
TYPE	string	Geometry type name

Note: The coord_dimension and srid properties are determined from the {http://www.teiid.org/translator/spatial/2015}coord_dimension and {http://www.teiid.org/translator/spatial/2015}srid extension properties on the column. When possible these values will be set automatically by the relevant importer. If they are not set, they will be reported as 2 and 0 respectively. If client logic expects actual values, such as integration with GeoServer, then you may need to set these values manually.

Procedures

SYS.getXMLSchemas

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

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

SYS.ArrayIterate

Returns a resultset with a single column with a row for each value in the array.

```
SYS.ArrayIterate(IN val object[]) RETURNS TABLE (col object)
```

Example ArrayIterate

```
select array_get(cast(x.col as string[]), 2) from (exec arrayiterate((('a', 'b'),('c','d')))) x
```

This will produce two rows - 'b', and 'd'.

SYSADMIN Schema

System schema for administrative information and actions.

Table of Contents

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

Tables/Views

SYSADMIN.Usage

This table supplies information about how views / procedures are defined.

Column Name	Type	Description
VDBName	string	VDB name
UID	string	Object UID
object_type	string	Type of object (StoredProcedure, View, etc.)
Name	string	Object Name or parent name if ElementName is not null
ElementName	string	Name of column or parameter
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 if Uses_ElementName is not null
Uses_ElementName	string	Used column or parameter name

Example SYSADMIN.Usage

```
SELECT * FROM SYSADMIN.Usage
```

Recursive common table queries can be used to determine transitive relationships.

Example Finding All Incoming Usage

```
with im_using as (
    select 0 as level, uid, Uses_UID, Uses_Name, Uses_Object_Type, Uses_ElementName
      from usage where uid = (select uid from sys.tables where name='table name' and schemaName='schema name')
    union all
    select level + 1, usage.uid, usage.Uses_UID, usage.Uses_Name, usage.Uses_Object_Type, usage.Uses_ElementNam
e
      from usage, im_using where level < 10 and usage.uid = im_using.Uses_UID) select * from im_using
```

Example Finding All Outgoing Usage

```
with uses_me as (
    select 0 as level, uid, Uses_UID, Name, Object_Type, ElementName
      from usage where uses_uid = (select uid from sys.tables where name='table name' and schemaName='schema na
me')
    union all
    select level + 1, usage.uid, usage.Uses_UID, usage.Name, usage.Object_Type, usage.ElementName
      from usage, uses_me where level < 10 and usage.uses_uid = uses_me.UID) select * from uses_me
```

SYSADMIN.MatViews

This table supplies information about all the materailized views in the virtual database.

Column Name	Type	Description
VDBName	string	VDB name
SchemaName	string	Schema Name
Name	string	Short group name
TargetSchemaName	string	Name of the materialized table schema. Will be null for internal materialization.
TargetName	string	Name of the materialized table
Valid	boolean	True if materialized table is currently valid. Will be null for external materialization.
LoadState	boolean	The load state, can be one of NEEDS_LOADING, LOADING, LOADED, FAILED_LOAD. Will be null for external materialization.

Updated	timestamp	The timestamp of the last full refresh. Will be null for external materialization.
Cardinality	integer	The number of rows in the materialized view table. Will be null for external materialization.

Valid, LoadState, Updated, and Cardinality may be checked for external materialized views with the SYSADMIN.matViewStatus procedure.

Example SYSADMIN.MatViews

```
SELECT * FROM SYSADMIN.MatViews
```

SYSADMIN.VDBResources

This table provides the current VDB contents.

Column Name	Type	Description
resourcePath	string	The path to the contents.
contents	blob	The contents as a blob.

Example SYSADMIN.VDBResources

```
SELECT * FROM SYSADMIN.VDBResources
```

SYSADMIN.Triggers

This table provides the Triggers in the virtual database.

Column Name	Type	Description
VDBName	string	VDB name
SchemaName	string	Schema Name
TableName	string	Table name
Name	string	Trigger name
TriggerType	string	Trigger Type
TriggerEvent	string	Triggering Event
Status	string	Is Enabled
Body	clob	Trigger Action (FOR EACH ROW …)
TableUID	string	Table Unique ID

Example SYSADMIN.Triggers

```
SELECT * FROM SYSADMIN.Triggers
```

SYSADMIN.Views

This table provides the Views in the virtual database.

Column Name	Type	Description
VDBName	string	VDB name
SchemaName	string	Schema Name
Name	string	View name
Body	clob	View Definition Body (SELECT …)
UID	string	Table Unique ID

Example SYSADMIN.Views

```
SELECT * FROM SYSADMIN.Views
```

SYSADMIN.StoredProcedures

This table provides the StoredProcedures in the virtual database.

Column Name	Type	Description
VDBName	string	VDB name
SchemaName	string	Schema Name
Name	string	Procedure name
Body	clob	Procedure Definition Body (BEGIN …)
UID	string	Unique ID

Example SYSADMIN.StoredProcedures

```
SELECT * FROM SYSADMIN.StoredProcedures
```

Procedures

SYSADMIN.isLoggable

Tests if logging is enabled at the given level and context.

```
SYSADMIN.isLoggable(OUT loggable boolean NOT NULL RESULT, IN level string NOT NULL DEFAULT 'DEBUG', IN context
string NOT NULL DEFAULT 'org.teiid.PROCESSOR')
```

Returns true if logging is enabled. level can be one of the log4j levels: OFF, FATAL, ERROR, WARN, INFO, DEBUG, TRACE. level defaults to 'DEBUG' and context defaults to 'org.teiid.PROCESSOR'

Example isLoggable

```
IF ((CALL SYSADMIN.isLoggable(context=>'org.something'))
BEGIN
   DECLARE STRING msg;
   // logic to build the message ...
   CALL SYSADMIN.logMsg(msg=>msg, context=>'org.something')
END
```

SYSADMIN.logMsg

Log a message to the underlying logging system.

```
SYSADMIN.logMsg(OUT logged boolean NOT NULL RESULT, IN level string NOT NULL DEFAULT 'DEBUG', IN context string
 NOT NULL DEFAULT 'org.teiid.PROCESSOR', IN msg object NOT NULL)
```

Returns true if the message was logged. level can be one of the log4j levels: OFF, FATAL, ERROR, WARN, INFO, DEBUG, TRACE. level defaults to 'DEBUG' and context defaults to 'org.teiid.PROCESSOR'

Example logMsg

```
CALL SYSADMIN.logMsg(msg=>'some debug', context=>'org.something')
```

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

SYSADMIN.refreshMatView

Full refresh/load of an internal materialized view. Returns integer RowsUpdated. -1 indicates a load is in progress, otherwise the cardinality of the table is returned. See the Caching Guide for more.

See also SYSADMIN.loadMatView

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

SYSADMIN.refreshMatViewRow

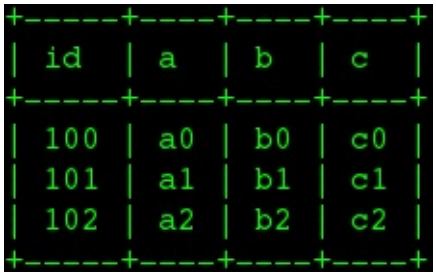
Refreshes a row in an internal materialized view.

Returns integer RowsUpdated. -1 indicates the materialized table is currently invalid. 0 indicates that the specified row did not exist in the live data query or in the materialized table. See the Caching Guide for more.

```
SYSADMIN.CREATE FOREIGN PROCEDURE refreshMatViewRow(OUT RowsUpdated integer NOT NULL RESULT, IN ViewName string
 NOT NULL, IN Key object NOT NULL, VARIADIC KeyOther object)
```

Example of SYSADMIN.refreshMatViewRow

The materialized view `SAMPLEMATVIEW` has 3 rows under the `TestMat` Model as below:



id	a	b	c
100	a0	b0	c0
101	a1	b1	c1
102	a2	b2	c2

Assuming the primary key only contains one column, id, update the second row:

```
EXEC SYSADMIN.refreshMatViewRow('TestMat.SAMPLEMATVIEW', '101')
```

Assuming the primary key contains more columns, a and b, update the second row:

```
EXEC SYSADMIN.refreshMatViewRow('TestMat.SAMPLEMATVIEW', '101', 'a1', 'b1')
```

SYSADMIN.refreshMatViewRows

Refreshes rows in an internal materialized view.

Returns integer RowsUpdated. -1 indicates the materialized table is currently invalid. Any row that does not exist in the live data query or in the materialized table will not count toward the RowsUpdated. See the Caching Guide for more.

```
SYSADMIN.refreshMatViewRows(OUT RowsUpdated integer NOT NULL RESULT, IN ViewName string NOT NULL, VARIADIC Key
object[] NOT NULL)
```

Example of SYSADMIN.refreshMatViewRows

Continuing use the `SAMPLEMATVIEW` in Example of SYSADMIN.refreshMatViewRow. Assuming the primary key only contains one column, id, update all rows:

```
EXEC SYSADMIN.refreshMatViewRows('TestMat.SAMPLEMATVIEW', ('100',), ('101',), ('102',))
```

Assuming the primary key comtain more columns, id, a and b compose of the primary key, update all rows:

```
EXEC SYSADMIN.refreshMatViewRows('TestMat.SAMPLEMATVIEW', ('100', 'a0', 'b0'), ('101', 'a1', 'b1'), ('102', 'a2
', 'b2'))
```

SYSADMIN.setColumnStats

Set statistics for the given column.

```
SYSADMIN.setColumnStats(IN tableName string NOT NULL, IN columnName string NOT NULL, IN distinctCount long, IN
nullCount long, IN max string, IN min string)
```

All stat values are nullable. Passing a null stat value will leave corresponding metadata value unchanged.

SYSADMIN.setProperty

Set an extension metadata property for the given record. Extension metadata is typically used by Translators.

```
SYSADMIN.setProperty(OUT OldValue clob NOT NULL RESULT, IN UID string NOT NULL, IN Name string NOT NULL, IN "Va
lue" clob)
```

Setting a value to null will remove the property.

Example Property Set

```
CALL SYSADMIN.setProperty(uid=>(SELECT uid FROM TABLES WHERE name='tab'), name=>'some name', value=>'some value'
)
```

This will set the property 'some name'='some value' on table tab.

Note	The use of this procedure will not trigger replanning of associated prepared plans.

Properties from built-in teiid_* namespaces can be set using the the short form - namespace:key form.

SYSADMIN.setTableStats

Set statistics for the given table.

```
SYSADMIN.setTableStats(IN tableName string NOT NULL, IN cardinality long NOT NULL)
```

Note	SYSADMIN.setColumnStats, SYSADMIN.setProperty, SYSADMIN.setTableStats are Metadata Procedures. A MetadataRepository must be configured to make a non-temporary metadata update persistent. See the Developer's Guide Runtime Metadata Updates section for more.

SYSADMIN.matViewStatus

matViewStatus is used to retrieve Materialized views' status via schemaName and viewName.

Returns tables which contains TargetSchemaName, TargetName, Valid, LoadState, Updated, Cardinality, LoadNumber, OnErrorAction.

```
SYSADMIN.matViewStatus(IN schemaName string NOT NULL, IN viewName string NOT NULL) RETURNS TABLE (TargetSchemaN
ame varchar(50), TargetName varchar(50), Valid boolean, LoadState varchar(25), Updated timestamp, Cardinality l
ong, LoadNumber long, OnErrorAction varchar(25))
```

SYSADMIN.loadMatView

loadMatView is used to perform a complete refresh of an internal or external materialized table.

Returns integer RowsInserted. -1 indicates the materialized table is currently loading. And -3 indicates there was an exception when performing the load. See the Caching Guide for more.

```
SYSADMIN.loadMatView(IN schemaName string NOT NULL, IN viewName string NOT NULL, IN invalidate boolean NOT NULL
 DEFAULT 'false') RETURNS integer
```

Example loadMatView

```
exec SYSADMIN.loadMatView(schemaName=>'TestMat',viewname=>'SAMPLEMATVIEW', invalidate=>'true')
```

SYSADMIN.updateMatView

The updateMatView procedure is used to update a subset of an internal or external materialized table based on the refresh criteria.

Returns integer RowsUpdated. -1 indicates the materialized table is currently invalid. And-3 indicates there was an exception when performing the update. See the Caching Guide for more.

```
SYSADMIN.updateMatView(IN schemaName string NOT NULL, IN viewName string NOT NULL, IN refreshCriteria string) R
ETURNS integer
```

SYSADMIN.updateMatView

Continuing use the `SAMPLEMATVIEW` in Example of SYSADMIN.refreshMatViewRow. Update view rows:

```
EXEC SYSADMIN.updateMatView('TestMat', 'SAMPLEMATVIEW', 'id = ''101'' AND a = ''a1''')
```

Translators

The Teiid Connector Architecture (TCA) provides Teiid with a robust mechanism for integrating with external systems. The TCA defines a common client interface between Teiid and an external system that includes metadata as to what SQL constructs are supported for pushdown and the ability to import metadata from the external system.

A Translator is the heart of the TCA and acts as the bridge logic between Teiid and an external system, which is most commonly accessed through a JCA resource adapter. Refer to the Teiid Developers Guide for details on developing custom Translators and JCA resource adapters for use with Teiid.

Tip	The TCA is not the same as the JCA, the JavaEE Connector Architecture, although the TCA is designed for use with JCA resource adapters.

Tip	The import capabilities of Teiid Translators can be utilized in Teiid Designer via the Teiid Connection Importer.

A Translator is typically paired with a particular JCA resource adapter. In instances where pooling, environment dependent configuration management, advanced security handling, etc. are not needed, then a JCA resource adapter is not needed. The configuration of JCA ConnectionFactories for needed resource adapters is not part of this guide, please see the Teiid Administrator Guide and the kit examples for configuring resource adapters for use in WildFly.

Translators can have a number of configurable properties. These are broken down into execution properties, which determine aspects of how data is retrieved, and import settings, which determine what metadata is read for import.

The execution properties for a translator typically have reasonable defaults. For specific translator types, e.g. the Derby translator, base execution properties are already tuned to match the source. In most cases the user will not need to adjust their values.

Table 1. **Base Execution Properties - shared by all translators**

Name	Description	Default
Immutable	Set to true to indicate that the source never changes.	false
RequiresCriteria	Set to true to indicate that source SELECT/UPDATE/DELETE queries require a where clause.	false
SupportsOrderBy	Set to true to indicate that the ORDER BY clause is supported.	false
SupportsOuterJoins	Set to true to indicate that OUTER JOINs are supported.	false
SupportsFullOuterJoins	If outer joins are supported, true indicates that FULL OUTER JOINs are supported.	false
SupportsInnerJoins	Set to true to indicate that INNER JOINs are supported.	false
SupportedJoinCriteria	If joins are supported, defines what criteria may be used as the join criteria. May be one of (ANY, THETA, EQUI, or KEY).	ANY

MaxInCriteriaSize	If in criteria are supported, defines what the maximum number of in entries are per predicate. -1 indicates no limit.	-1
MaxDependentInPredicates	If in criteria are supported, defines what the maximum number of predicates that can be used for a dependent join. Values less than 1 indicate to use only one in predicate per dependent value pushed (which matches the pre-7.4 behavior).	-1
DirectQueryProcedureName	if the direct query procedure is supported on the translator, this property indicates the name of the procedure.	native
SupportsDirectQueryProcedure	Set to true to indicate the translator supports the direct execution of commands	false
ThreadBound	Set to true to indicate the translator's Executions should be processed by only a single thread	false
CopyLobs	If true, then returned lobs (clob, blob, sql/xml) will be copied by the engine in a memory safe manner. Use this option if the source does not support memory safe lobs or you want to disconnect lobs from the source connection.	false
TransactionSupport	The highest level of transaction support. Used by the engine as a hint to determine if a transaction is needed for autoCommitTxn=DETECT mode. Can be one of XA, NONE, or LOCAL. If XA, then access under a transaction will be serialized.	XA

Note	Only a subset of the supports metadata can be set through execution properties. If more control is needed, please consult the Developer's Guide.

There are no base importer settings.

Override Execution Properties

For all the translators to override Execution Properties can be configured in the *vdb.xml* file.

Example Overriding of Translator Property

```
<model name="ora">
     <source name="ora" translator-name="oracle-override" connection-jndi-name="java:/oracle"/>
</model>

<translator name="oracle-override" type="oracle">
     <property name="RequiresCriteria" value="true"/>
```

```
  </translator>
```

The above XML fragment is overriding the *oracle* translator and altering the behavior of *RequiresCriteria* property to true. Note that the modified translator is only available in the scope of this VDB.

See also VDB Definition.

Parameterizable Native Queries

In some situations the teiid_rel:native-query property and native procedures accept parameterizable strings that can positionally reference IN parameters. A parameter reference has the form $integer, i.e. $1 Note that 1 based indexing is used and that only IN parameters may be referenced. Dollar-sign integer is therefore reserved, but may be escaped with another $, i.e. $$1. The value will be bound as a prepared value or a literal is a source specific manner. The native query must return a result set that matches the expectation of the calling procedure.

For example the native-query `select c from g where c1 = $1 and c2 = '$$1'` results in a JDBC source query of `select c from g where c1 = ? and c2 = '$1'`, where ? will be replaced with the actual value bound to parameter 1.

Amazon SimpleDB Translator

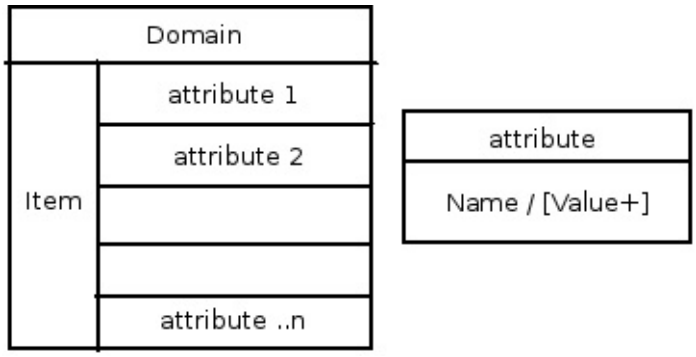
The Amazon SimpleDB Translator, known by the type name *simpledb*, exposes querying functionality to Amazon SimpleDB Data Sources.

Note	**"Amazon SimpleDB"** - Amazon SimpleDB is a web service for running queries on structured data in real time. This service works in close conjunction with Amazon Simple Storage Service (Amazon S3) and Amazon Elastic Compute Cloud (Amazon EC2), collectively providing the ability to store, process and query data sets in the cloud. These services are designed to make web-scale computing easier and more cost-effective for developers. Read more about it at http://aws.amazon.com/simpledb/

This translator provides an easy way connect to Amazon SimpleDB and provides relational way using SQL to add records from directly from user or from other sources that are integrated with Teiid. It also gives ability to read/update/delete existing records from SimpleDB store.

Usage

Amazon SimpleDB is hosted key/value store where a single key can contain host multiple attribute name/value pairs where where value can also be a multi-value. The data structure can be represented by



Based on above data structure, when you import the metadata from SimpleDB into Teiid, the constructs are aligned as below

Simple DB Name	SQL (Teiid)
Domain	Table
Item Name	Column (ItemName) Primary Key
attribute - single value	Column - String Datatype
attribute - multi value	Column - String Array Datatype

Since all attributes are by default are considered as string data types, columns are defined with string data type. However, during modeling of the schema in Designer, one can use various other data types supported through Teiid to define a data type of column, that user wishes to expose as.

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"

If you are using Designer Tooling, to create VDB

Create/use a Teiid Designer Model project

Use "Teiid Connection >> Source Model" importer, create SimpleDB Data Source using data source creation wizard and use *simpledb* as translator in the importer. The table(s) is created in a source model by the time you finish with this importer, if the data is already defined on Amazon SimpleDB.

Create a VDB and deploy into Teiid Server and use either jdbc, odbc, odata etc to query.

Properties

The Amazon SimpleDB Translator currently has no import or execution properties.

Capabilities

The Amazon SimpleDB Translator supports SELECT statements with a restrictive set of capabilities including: comparison predicates, IN predicates, LIMIT and ORDER BY. Insert, update, delete are also supported.

Queries on Attributes with Multiple Values

Attributes with multiple values will defined as string array type. So this column is treated SQL Array type. The below table shows SimpleDB way of querying to Teiid way to query. The queries are based on http://docs.aws.amazon.com/AmazonSimpleDB/latest/DeveloperGuide/RangeValueQueriesSelect.html

SimpleDB Query	Teiid Query
select * from mydomain where Rating = '4 stars' or Rating = '**'	select * from mydomain where Rating = ('4 stars','**')
select * from mydomain where Keyword = 'Book' and Keyword = 'Hardcover'	select * from mydomain where intersection(Keyword,'Book','Hardcover')
select * from mydomain where every(Rating) = '**'	select * from mydomain where every(Rating) = '**'

With Insert/Update/Delete you write prepare statements or you can write SQL like

```
INSERT INTO mydomain (ItemName, title, author, year, pages, keyword, rating) values ('0385333498', 'The Sirens of Titan', 'Kurt Vonnegut', ('1959'), ('Book', Paperback'), ('*****','5 stars','Excellent'))
```

Direct Query Support

	This feature is turned off by default because of the security risk this exposes to execute any command against the

	source. To enable this feature, override the execution property called SupportsDirectQueryProcedure to true.

Tip	By default the name of the procedure that executes the queries directly is called native. Override the execution property DirectQueryProcedureName to change it to another name.

The SimpleDB translator provides a procedure to execute any ad-hoc simpledb query directly against the source without Teiid parsing or resolving. Since the metadata of this procedure's results are not known to Teiid, they are returned as an object array. ARRAYTABLE can be used construct tabular output for consumption by client applications. Direct query supported for "select" based calls.

```
SELECT X.*
  FROM simpledb_source.native('SELECT firstname, lastname FROM users') n, ARRAYTABLE(n.tuple COLUMNS firstname
string, lastname string) AS X
```

JCA Resource Adapter

The Teiid specific Amazon SimpleDB Resource Adapter should be used with this translator. See Amazon SimpleDB Data Sources for connecting to SimpleDB.

Apache Accumulo Translator

The Apache Accumulo Translator, known by the type name *accumulo*, exposes querying functionality to Accumulo Data Sources. Apache Accumulo is a sorted, distributed key value store with robust, scalable, high performance data storage and retrieval system. This translator provides an easy way connect to Accumulo system and provides relational way using SQL to add records from directly from user or from other sources that are integrated with Teiid. It also gives ability to read/update/delete existing records from Accumulo store. Teiid has capability to pass-in logged in user's roles as visibility properties to restrict the data access.

Tip	**"versions"** - The development was done using Accumulo 1.5.0, Hadoop 2.2.0 and Zookeeper 3.4.5

Note	This document assumes that user is familiar with Accumulo source and has basic understanding of how Teiid works. This document only contains details about Accumulo translator.

Intended Usecases

The usage Accumulo translator can be highly dependent on user's usecase(s). Here are some common scenarios.

- Accumulo source can be used in Teiid, to continually add/update the documents in the Accumulo system from other sources automatically.
- Access Accumulo through SQL interface.
- Make use of cell level security through enterprise roles.
- Accumulo translator can be used as an indexing system to gather data from other enterprise sources such as RDBMS, Web Service, SalesForce etc, all in single client call transparently with out any coding.

Usage

Apache Accumulo is distributed key value store with unique data model. It allows to group its key-value pairs in a collection called "table". The key structure is defined as

<table>
<tr><th colspan="4">Key</th><th rowspan="2">Value</th></tr>
<tr><th rowspan="2">Row ID</th><th colspan="3">Column</th><th rowspan="2">Timestamp</th></tr>
<tr><th>Family</th><th>Qualifier</th><th>Visibility</th></tr>
</table>

Based on above information, one can define a schema representing Accumulo table structures in Teiid using DDL or using Teiid Designer 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"

If you are using Designer Tooling, to create VDB

- Create/use a Teiid Designer Model project
- Use "Teiid Connection >> Source Model" importer, create Accumulo Data Source using data source creation wizard and use *accumulo* as translator in the importer. The table is created in a source model by the time you finish with this importer.
- Create a VDB and deploy into Teiid Server and use either jdbc, odbc, odata etc to query.

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 or using the Teiid Designer. 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

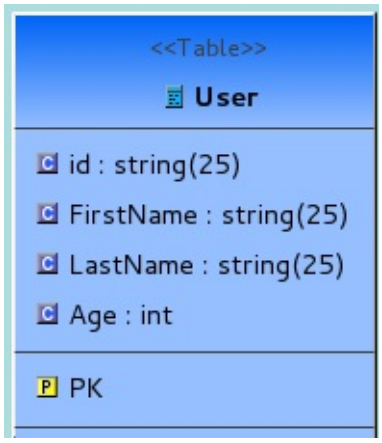
```
CREATE FOREIGN TABLE "User" (
    rowid string OPTIONS (UPDATABLE FALSE, SEARCHABLE 'All_Except_Like'),
    name_age string OPTIONS (SEARCHABLE 'All_Except_Like', "teiid_accumulo:CF" 'name', "teiid_accumulo:CQ" 'age'
, "teiid_accumulo:VALUE-IN" '{VALUE}'),
    name_firstname string OPTIONS (SEARCHABLE '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.

If you did not use built in import, and would like to manually design the table in Designer like below



Then you must make sure you supply the Extension Metadata Properties defined above on the User table's columns from Accumulo extended metadata(In Designer, right click on Model, and select "Model Extension Definitions" and select Accumulo. For example on FirstName column, you would supply

```
teiid_accumulo:CF  name
teiid_accumulo:CQ  firstname
teiid_accumulo:VALUE-IN  VALUE
```

and repeat for each and every column, so that Teiid knows how to communicate correctly with Accumulo.

JCA Resource Adapter

The Teiid specific Accumulo Resource Adapter should be used with this translator. See Accumulo Data Sources for connecting to a Accumulo Source.

Native Queries

Currently this feature is not applicable. Based on user demand Teiid could expose a way for user to submit a MAP-REDUCE job.

Direct Query Procedure

This feature is not applicable for this translator.

Apache SOLR Translator

The Apache SOLR Translator, known by the type name *solr*, exposes querying functionality to Solr Data Sources. Apache Solr is a search engine built on top of Apache Lucene for indexing and searching. This translator provides an easy way connect to existing or a new Solr search system, and provides way to add documents/records from directly from user or from other sources that are integrated with Teiid. It also gives ability to read/update/delete existing documents from Solr Search system.

Properties

The Solr Translator currently has no import or execution properties. It does not define any extension metadata.

Intended Usecases

The usage Solr translator can be highly dependent on user's usecase(s). Here are some common scenarios.

- Solr source can be used in Teiid, to continually add/update the documents in the search system from other sources automatically.
- If the search fields are stored in Solr system, this can be used as very low latency data retrieval for serving high traffic applications.
- Solr translator can be used as a fast full text search. The Solr document can contain only the index information, then the results as an inverted index to gather target full documents from the other enterprise sources such as RDBMS, Web Service, SalesForce etc, all in single client call transparently with out any coding.

Usage

Solr search system provides searches based on indexed search fields. Each Solr instance is typically configured with a single core that defines multiple fields with different type information. Teiid metadata querying mechanism is equipped with "Luke" based queries, that at deploy time of the VDB use this mechanism to retrieve all the stored/indexed fields. Currently Teiid does NOT support dynamic fields and non-stored fields. Based on retrieved fields, Solr translator exposes a single table that contains all the fields. If a field is multi-value based, it's type is represented as Array type.

Once this table is exposed through VDB in a Teiid database, and Solr Data Sources is created, the user can issue "INSERT/UPDATE/DELETE" based SQL calls to insert/update/delete documents into the Solr, and issue "SELECT" based calls to retrieve documents from Solr. You can use full range of SQL with Teiid system integrating other sources along with Solr source.

The Solr Translator supports SELECT statements with a restrictive set of capabilities including: comparison predicates, IN predicates, LIMIT and Order By.

An Example VDB that shows Solr translator can be defined as

```
<vdb name="search" version="1">
    <model name="solr">
        <source name="node-one" translator-name="solr" connection-jndi-name="java:/solrDS"/>
    </model>
<vdb>
```

The translator does NOT provide a connection to the Solr. For that purpose, Teiid has a JCA adapter that provides a connection to Solr using the SolrJ Java library. To define such connector, see Solr Data Sources or see an example in "<jboss-as>/docs/teiid/datasources/solr"

If you are using Designer Tooling, to create VDB then

- Create/use a Teiid Designer Model project
- Use "Teiid Connection >> Source Model" importer, create Solr Data Source using data source creation wizard and use *solr* as translator in the importer. The search table is created in a source model by the time you finish with this importer.
- Create a VDB and deploy into Teiid Server and use either jdbc, odbc, odata etc to query.

JCA Resource Adapter

The Teiid specific Solr Resource Adapter should be used with this translator. See Solr Data Sources for connecting to a Solr Search Engine.

Native Queries

This feature is not applicable for Solr translator.

Direct Query Procedure

This feature is not available for Solr translator currently.

Cassandra Translator

The Cassandra Translator, known by the type name *cassandra*, exposes querying functionality to Cassandra Data Sources. The translator translates Teiid push down commands into Cassandra CQL.

Properties

The Cassandra Translator currently has no import or execution properties.

Usage

The Cassandra Translator supports INSERT/UPDATE/DELETE/SELECT statements with a restrictive set of capabilities including: count(*), comparison predicates, IN predicates, and LIMIT. Only indexed columns are searchable. Consider a custom extension or create an enhancement request should your usage require additional capabilities.

If you are using Designer Tooling, to create VDB then:

- Create/use a Teiid Designer Model project
- Use "Teiid Connection >> Source Model" importer, create a new JBoss Data Source connection profile, specifying the JNDI name for resource adapter configured https://docs.jboss.org/author/display/TEIID Cassandra+Data+Sources[data source] and use *cassandra* as translator type. The source model will be created when you finish with this importer.
- Create a VDB and deploy into Teiid Server and use either jdbc, odbc, odata etc to query.

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
```

Delegating Translators

Translator "delegator"

A translator by name "delegator" is available in core Teiid installation, that can be used to modify the capabilities of a existing translator. Often times for debugging purposes or in special situations, one may require to either turn on/off certain capability of translator. For example, assume Hive database in their latest version supporting the ORDER BY construct, however Teiid's current version of the Hive translator does not have this capability, you can use the "delegator" translator to turn ON the "ORDER BY" support without actually writing any code. Sometimes you may want to do the reverse, you want turn off certain capability to produce a better plan. In these situations, you can use this translator.

To use this translator, you need to define this translator in the VDB, as a shown in the below VDB. The below example overriding the "hive" translator and turning off the ORDER BY support.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<vdb name="myvdb" version="1">

    <model name="mymodel">
        <source name="source" translator-name="hive-delegator" connection-jndi-name="java:hive-ds"/>
    </model>

    <!-- the below it is called translator overriding, where you can set different properties -->
    <translator name="hive-delegator" type="delegator" />
        <property name="delegateName" value="hive" />
        <property name="supportsOrderBy" value="false"/>
   </translator>
</vdb>
```

You can override any/all the translator capabilities defined here Translator Capabilities as Execution Properties to override. Example of "supportsOrderBy" is shown in above example.

Extending the "delegator" translator

You may create a delegating translator by extending the `org.teiid.translator.BaseDelegatingExecutionFactory` . Once your classes are then packaged as a custom translator, you will be able to wire another translator instance into your delegating translator at runtime in order to intercept all of the calls to the delegate. This base class does not provide any functionality on its own, other than delegation. The difference here from previous "delegator" translator is, you can hard code the capabilities instead of defining as configuration inside the -vdb.xml, as well as override methods to provide alternate behavior.

Execution Properties

Name	Description	Default
delegateName	Translator instance name to delegate to	n/a

Lets say you are currently using "oracle" translator in your VDB, you want to intercept the calls going through this translator, then you first write a custom delegating translator like

```
@Translator(name="interceptor", description="interceptor")
public class InterceptorExecutionFactory extends org.teiid.translator.BaseDelegatingExecutionFactory{
    @Override
    public void getMetadata(MetadataFactory metadataFactory, C conn) throws TranslatorException {
```

```
        // do intercepting code here..

        // If you want call the original delegate, do not call if do not need to.
        // but if you did not call the delegate fullfill the method contract
        super.getMetadata(metadataFactory, conn);

        // do more intercepting code here..
    }
}
```

Now deploy this translator in Teiid engine. Then in your -vdb.xml or .vdb file define like below.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<vdb name="myvdb" version="1">

    <model name="mymodel">
        <source name="source" translator-name="oracle-interceptor" connection-jndi-name="java:oracle-ds"/>
    </model>

    <!-- the below it is called translator overriding, where you can set different properties -->
    <translator name="orcle-interceptor" type="interceptor" />
        <property name="delegateName" value="oracle" />
   </translator>
</vdb>
```

We have defined a "translator" override called "oracle-interceptor", which is based on the custom translator "interceptor" from above, and supplied the translator it needs to delegate to "oracle" as its delegateName. Then, we used this override translator "oracle-interceptor" in your VDB. Now any calls going into this VDB model’s translator will be intercepted by YOUR code to do whatever you want to do.

File Translator

The file translator, known by the type name *file*, exposes stored procedures to leverage file system resources exposed by the File Data Source and the FTP Data Source. It will commonly be used with the TEXTTABLE or XMLTABLE table functions to use CSV or XML formatted data.

Execution Properties

Name	Description	Default
Encoding	The encoding that should be used for CLOBs returned by the getTextFiles procedure	The system default encoding
ExceptionIfFileNotFound	Throw an exception in getFiles or getTextFiles if the specified file/directory does not exist.	true (false prior to 8.2)

Usage

getFiles

Retrieve all files as BLOBs with an optional extension at the given path.

```
call getFiles('path/*.ext')
```

If the path is a directory, then all files in the directory will be returned. If the path matches a single file, it will be returned.

The '*' character will be treated as a wildcard to match any number of characters in the path name - 0 or matching files will be returned.

If '*' is not used and the path doesn’t exist and ExceptionIfFileNotFound is true, then an exception will be raised.

getTextFiles

Retrieve all files as CLOB(s) with the an optional extension at the given path.

```
call getTextFiles('path/*.ext')
```

All the same files a getFiles will be retrieved, the only difference is that the results will be CLOB values using the encoding execution property as the character set.

saveFile

Save the CLOB, BLOB, or XML value to given path

```
call saveFile('path', value)
```

deleteFile

Delete the file at the given path

```
call deleteFile('path')
```

The path should reference an existing file. If the file does not exist and ExceptionIfFileNotFound is true, then an exception will be thrown. Or if the file cannot be deleted an exception will be thrown.

NOTE **Native queries -** Native or direct query execution is not supported on the File Translator.

JCA Resource Adapter

The resource adapter for this translator provided through "File Data Source", Refer to Admin Guide for configuration information.

Google Spreadsheet Translator

The *google-spreadsheet* translator is used to connect to a Google Spreadsheet. To use the Google Spreadsheet Translator you need to configure and deploy the Google JCA connector - see the Admin Guide.

The query approach expects the data in the worksheet to be in a specific format. Namely:

- Any column that has data is queryable.
- Any column with an empty cell has the value retrieved as null. However differentiating between null string and empty string values may not always be possible as google treats them interchanably. Where possible the translator may provide a warning or throw an exception if there may be a confusion of null vs. empty strings.
- If the first row is present and contains string values, then it will be assumed to represent the column labels.

If you are using the default native metadata import, the metadata for your Google account (worksheets and information about columns in worksheets) are loaded upon translator start up. If you make any changes in data types, it is advisable to restart your vdb.

The translator supports queries against a single sheet. It supports ordering, aggregation, basic predicates, and most of the functions supported by the spreadsheet query language.

There are no google-spreadsheet importer settings, but it can provide metadata for VDBs.

JCA Resource Adapter

The Teiid specific Google Spreadsheet Data Sources Resource Adapter should be used with this translator.

Native Queries

Google spreadsheet source procedures may be created using the teiid_rel:native-query extension - see Parameterizable Native Queries. The procedure will invoke the native-query similar to an native procedure call with the benefits that the query is predetermined and that result column types are known, rather than requiring the use of ARRAYTABLE or similar functionality. See the Select format below.

Direct Query Procedure

Note	This feature is turned off by default because of the security risk this exposes to execute any command against the source. To enable this feature, override the execution property called _SupportsDirectQueryProcedure to true.

Tip	By default the name of the procedure that executes the queries directly is called **native**. Override the execution property _DirectQueryProcedureName to change it to another name.

The Google spreadsheet translator provides a procedure to execute any ad-hoc query directly against the source without any Teiid parsing or resolving. Since the metadata of this procedure's execution results are not known to Teiid, they are returned as an object array. ARRAYTABLE can be used construct tabular output for consumption by client applications. Teiid exposes this procedure with a simple query structure as below:

Select

Select Example

```
SELECT x.* FROM (call google_source.native('worksheet=People;query=SELECT A, B, C')) w,
```

```
ARRAYTABLE(w.tuple COLUMNS "id" string , "type" string, "name" String) AS x
```

the first argument takes semi-colon(;) separated name value pairs of following properties to execute the procedure:

Property	Description	Required
worksheet	Google spreadsheet name	yes
query	spreadsheet query	yes
limit	number rows to fetch	no
offset	offset of rows to fetch from limit or beginning	no

Infinispan HotRod Translator

The Infinispan HotRod Translator, known by the type **ispn-hotrod**, can read the java objects from a remote JDG Cache via the Hot Rod client using the Google Protobuf for serialization. This will enable Teiid to query the remote cache using JDG DSL. This translator extends the Object Translator and uses it for the core processing of reading and writing objects.

If you are using JDG in library mode, see the Infinispan Library Mode Translator for this type of configuration.

Usage

- Retrieve objects from a cache and transform into rows and columns.
- Supports performing writes to the cache
- Use for external materialization to improve query performance

Supported Capabilities

The following are the connector capabilities:

- Compare Criteria - EQ
- Compare Criteria Ordered - LT, GT, LE, GE - support for SupportsComapareCriteriaOrdered will be controlled by the version of JDG being accessed. Any JDG version 6.5 and prior will have this set to false due to an issue with JDG.
- And/Or Criteria
- In Criteria
- Like Criteria
- Order By
- INSERT, UPDATE, DELETE (non-transactional)

The following will not be pushed down to JDG for processing, but will be done within Teiid:

- Not (NE, <>)
- IsNull

Limitations:

- support for 'Not' has been disabled, because JDG considers <column when null> <> 1 to be true and SQL does not.
- boolean data type: JDG will throw an exception if no value is specified on the insert or when no default value is defined in the protobuf definition file.
- char data type: is not a supported type in theProtobuf data types (https://developers.google.com/protocol-buffers/docs/proto#scalar). Would either have to handle conversion in the protobuf marshaller or create a Teiid view with the data type as char.
- 1-to-Many, currently only supports Collection or Array, not Map’s'
- XA transactions are not supported by JDG when using Hot Rod client.

To be done:

- support deleting containing class(s) (i.e., person -→ phones, delete a phone)
- support updating containing class(s)

Installation

The **ispn-hotrod** translator is not configured, out-of-the-box, in the standalone-teiid.xml configuration. To configure the translator, run the add-infinispan-hotrod-translator.cli script. This script can also be found in the teiid-jboss-dist.zip kit, under docs/teiid/datasources/infinispan. See the jdg-remote-cache quick start for an example. Note, this assumes you will be installing a separate JDG server, which will be accessed via the Hot Rod Client installed into the Teiid server.

Metadata

Options for Defining

There are several options to defining the metadata representing your object in the cache.

- "Recommended" Use the Teiid Connection Importer in Teiid Designer to create the physical source model based on your object cache. The table columns will be created from the google protobuf definition, that corresponds to a registered class.
- Use Teiid Designer to manually create the physical source model based on your object cache using the below **Definition Requirements**.
- A simple VDB that only defines the data source to use. Example:

```
    <model name="People" type="Physical">
        <property name="importer.useFullSchemaName" value="false"/>

        <source name="infinispan-hotrod-connector" translator-name="ispn-hotrod" connection-jndi-name="java:/in
finispanRemoteDSL" />
    </model>
```

The metadata will be resolved by reverse engineering the defined object in the cache. This can be useful when using the Teiid Designer Teiid Connection Importer for building the physical source model(s).

- You can also define the metadata using DDL. See Object Translator for an example.

Definition Requirements

- see Object Translator Metadata section for base definition requirements.
- Columns will be identified as SEARCHABLE if either the protobuf definition for a column indicates its indexed or the pojo class has the attribute/method annotated.
- A 1-to-* relationship class must have a foreign key to map to the root class/table, where the name in source for the foreign key is the name of the root class method to access those child objects. Note, this is the class method, not a reference in the google protobuf definition.
- A container/child class will have attributes where the NIS contain a period. Example: phone.number. This is because this maps to to google protobuf definition and what is expected by the DSL query to reference attributes of a inner class.

External Materialization

This translator supports using the cache for external materialization. However, there are specific configuration settings that are required on the JDG HotRod Resource Adapter and at the translator.

Note	Creating or selecting a view to materialize: Unfortunately for protobufs, complex data types don't convert to C / C++. So when you are creating a view to use for materialization, it is recommended to stick with primitive data types because protobuffer doesn't support bigdecimal directly. So there are 3 options: 1. use all primitive data types 2. implement a marshaller that will handle the conversion, which means the .proto file will also need to be created (see JDG for creation of files) 3. create a view that will convert the bigdecimal to a string, then materialize that view.

Native Queries

External materialization is enabled by the use of native queries in the BEFORE_LOAD_SCRIPT and AFTER_LOAD_SCRIPT. A translator override will need to be set to enable native queries: SupportsNativeQueries=true

The following materialization properties must be defined:

Script	Native query	Description
teiid_rel:MATVIEW_BEFORE_LOAD_SCRIPT	truncate cache	To truncate the cache identified as the staging cache
teiid_rel:MATVIEW_AFTER_LOAD_SCRIPT	swap cache names	To swap the aliases for the caches, so that the primary cache points to the recently loaded cache

The following is an example of for defining the load scripts in DDL:

```
..
"teiid_rel:MATVIEW_BEFORE_LOAD_SCRIPT" 'execute StockMatCache.native(''truncate cache'');',
"teiid_rel:MATVIEW_LOAD_SCRIPT" 'insert into StockMatCache.Stock (productId, symbol, price, companyName) SELECT
  A.ID, S.symbol, S.price, A.COMPANY_NAME FROM Stocks.StockPrices AS S, Accounts.PRODUCT AS A WHERE S.symbol =
A.SYMBOL',
"teiid_rel:MATVIEW_AFTER_LOAD_SCRIPT"  'execute StockMatCache.native(''swap cache names'');',
```

Native queries are used to simulate how its done using RDBMS and renaming tables, because Infinispan doesn't currently support renaming a cache. So the native queries will trigger the clearing of the "staging" cache, and the swapping of the cache aliases.

Direct Query Procedure

Additionally, the execution of native queries is done thru the support of direct query procedures. The procedure to be executed is called **native**.

Warning	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 [Override Execution Properties] called **SupportsDirectQueryProcedure** to true.

Metadata Requirements

If you manually model the cache table in Teiid Designer, then you will need to add the property extension for defining the property "primary_table". The following is a DDL example:

```
SET NAMESPACE 'http://www.teiid.org/translator/object/2016' AS n0;

CREATE FOREIGN TABLE Trade (
         ....
        CONSTRAINT PK_TRADEID PRIMARY KEY(tradeId)
) OPTIONS (UPDATABLE TRUE);

CREATE FOREIGN TABLE ST_Trade (
        ....
) OPTIONS (NAMEINSOURCE 'Trade', UPDATABLE TRUE, "n0:primary_table" 'ObjectSchema.Trade');
```

JCA Resource Adapter

See JDG HotRod Data Sources resource adapter for this translator. It can be configured to lookup the cache container via JNDI, server list, or hot rod properties.

Infinispan Library Mode Translator

The Infinispan Library Mode translator, known by the name of **ispn-lib-mode**, is a bridge for reading and writing java objects to/from an JDG Cache running in library mode. This translator extends the Object Translator and uses it for the core processing of reading and writing objects. The Infinispan Library Mode Translator is written so that it can control how the cache is searched and any capabilities that are needed to control that behavior.

If you are using JDG to access a remote cache, see the Infinispan HotRod Translator for this type of configuration.

Usage

- Retrieve objects from a cache and transform into rows and columns.
- Perform writes to the cache
- Use for external materialization to improve query performance

Searching Options

Supports the following options for searching:

- DSL searching (JDG 6.3+)
- Hibernate/Lucene searching (now deprecated)
- Key searching (when no indexing is used).

See the jdg-local-cache quick start for an example.

Supported Capabilities

The following are the connector capabilities when Key Searching is used:

- SELECT command
- CompareCriteria - only EQ
- InCriteria

The following are the connector capabilities when DSL Searching is enabled:

- SELECT command
- CompareCriteria - EQ, NE
- CompareCriteria - GT, LT, GE, and LE can be enabled, see Execution Properties
- And/Or Criteria
- In Criteria
- Like Criteria
- Or Criteria
- IsNull check - see Execution Properties
- Not (negation) - example: Not In, Not Like

- INSERT, UPDATE, DELETE

Installation

The **ispn-lib-mode** translator is not configured, out-of-the-box, in the standalone-teiid.xml configuration. To configure the translator, run the add-infinispan-libmode-translator.cli script. This script can also be found in the teiid-jboss-dist.zip kit, under docs/teiid/datasources/infinispan.

Execution Properties

Name	Description	Required	Default
SupportsLuceneSearching (Deprecated)	Setting to true assumes your objects are annotated and Hibernate/Lucene will be used to search the cache	No	false
SupportsDSLSearching	Setting to true assumes your are using JDG v6.3 or better and your cache has indexing enabled	No	true
SupportsIsNullCriteria	Setting to true assumes https://issues.jboss.org/browse/TEIID-3539 has been resolved	No	false
SupportsCompareCriteriaOrdered	Setting to true assumes https://issues.jboss.org/browse/TEIID-3627 has been resolved	No	false
SupportsNotCriteria	Setting to true assumes https://issues.jboss.org/browse/TEIID-3573 has been resolved	No	false

SupportsNotCriteria defaults to false because the testing of *column* <> 1 returns true when the *column* is null, which isn't correct by SQL standards - see https://issues.jboss.org/browse/TEIID-3573. There is an enhancement coming that will enable adding "*column* IS NOT NULL" when *column* <> 1 type criteria is detected.

SupportsIsNullCriteria defaults to false because Infinispan/JDG has an issue with issuing a IS NULL check on a numeric type attribute - see https://issues.jboss.org/browse/TEIID-3539. Set this to true if you need this check and can control which non-numeric columns that this will be issued against.

SupportsCompareCriteria Ordered defaults to false because the Infinispan/JDG has an issue with string comparisons - see https://issues.jboss.org/browse/TEIID-3627.

Configuring Pojo class

The pojo class is the object that will be used to store the data in the cache. It should be built accordingly:

- To take advantage of the cache being indexed enabled, should annotate the class. See JDG documentation at https://access.redhat.com/documentation/en-US/Red_Hat_JBoss_Data_Grid/6.6/html-single/Infinispan_Query_Guide/index.html#chap-Annotating_Objects_and_Querying
- The class should be packaged into a jar so that it can be deployed as a module

Example class

```
@Indexed
public class Stock {
```

```
        @NumericField @Field(index=Index.YES, store=Store.YES, analyze=Analyze.NO)
        public int productId;
        @Field(index=Index.YES, store=Store.YES, analyze=Analyze.NO)
        public  BigDecimal price;
        @Field(index=Index.YES, store=Store.YES, analyze=Analyze.NO)
        public  String symbol;
        @Field(index=Index.YES, store=Store.YES, analyze=Analyze.NO)
        public  String companyName;

        public int getProductId() {
                return this.productId;
        }

        public void setProductId(int id) {
                this.productId = id;
        }

        public BigDecimal getPrice() {
                return this.price;
        }

        public void setPrice(BigDecimal price) {
                this.price = price;
        }
        public void setPrice(double price) {
                this.price = new BigDecimal(price);
        }

        public String getSymbol() {
                return this.symbol;
        }

        public void setSymbol(String symbol) {
                this.symbol = symbol;
        }

        public String getCompanyName() {
                return companyName;
        }

        public void setCompanyName(String name) {
                this.companyName = name;
        }
}
```

To configure the use of the pojo, do the following:

- Deploy the pojo jar as a module in the jboss-as server. Then define the "lib" property in the -vdb.xml and assign the correct module name. This can be done using the following template:

```
    <property name ="lib" value ="{pojo_module_name}"></property>
```

- The JDG commons module, org.infinispan.commons, slot="jdg-6.6" or slot for version installed, needs to have the pojo dependency added:

```
    <module name="{pojo_module_name}"   export="true" />
```

Metadata

Options for Defining

There are couple options to defining the metadata representing your object in the cache.

- "Recommended" Use the Teiid Connection Importer in Teiid Designer to create the physical source model based on your object cache. The table columns will be created from the google protobuf definition, that corresponds to a registered class.
- Use Teiid Designer to manually create the physical source model based on your object cache using the below **Definition Requirements**.

The following is a VDB example similar to quick start (see github at https://github.com/teiid/teiid-quickstarts/tree/master/jdg-local-cache). It defines the physical source and will use the translator native importer logic to have the metadata reverse engineered from the Stock class, see above, to be exposed as the source table.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<vdb name="Stocks" version="1">

    <description>Shows how to call an Infinispan local cache</description>

    <property name="cache-metadata" value="true" />
    <property name ="lib" value ="com.client.quickstart.pojos"></property>

    <model name="StockCache" type="Physical">
        <property name="importer.useFullSchemaName" value="false"/>

       <source name="StockSource" translator-name="infinispan1" connection-jndi-name="java:/infinispanLocal" />

    </model>

    <translator name="infinispan1" type="ispn-lib-mode">
        <property name="SupportsDSLSearching" value="true"/>
    </translator>
</vdb>
```

The metadata will be resolved by reverse engineering the defined object in the cache. This can be useful when using the Teiid Designer Teiid Connection Importer for building the physical source model(s).

- You can also define the metadata using DDL. See Object Translator for an example.

Definition Requirements

- see Object Translator Metadata section for base definition requirements.
- All columns that are not the primary key nor covered by a lucene index should be marked as SEARCHABLE 'Unsearchable'.

External Materialization

This translator supports using the cache for external materialization. However, there are specific configuration changes that are required at the Infinispan Data Sources resource-adapter and at the translator. For an example, see the jdg local quick start.

Native Queries

External materialization is enabled by the use of native queries in the BEFORE_LOAD_SCRIPT and AFTER_LOAD_SCRIPT. A translator override will need to be set to enable native queries: SupportsNativeQueries=true

The following materialization properties must be defined:

Script	Native query	Description
teiid_rel:MATVIEW_BEFORE_LOAD_SCRIPT	truncate cache	To truncate the cache identified as the staging cache

teiid_rel:MATVIEW_AFTER_LOAD_SCRIPT	swap cache names	To swap the aliases for the caches, so that the primary cache points to the recently loaded cache

The following is an example of defining the materialization load scripts in DDL:

```
..
"teiid_rel:MATVIEW_BEFORE_LOAD_SCRIPT" 'execute StockMatCache.native(''truncate cache'');',
"teiid_rel:MATVIEW_LOAD_SCRIPT" 'insert into StockMatCache.Stock (productId, symbol, price, companyName) SELECT
  A.ID, S.symbol, S.price, A.COMPANY_NAME FROM Stocks.StockPrices AS S, Accounts.PRODUCT AS A WHERE S.symbol =
A.SYMBOL',
"teiid_rel:MATVIEW_AFTER_LOAD_SCRIPT"  'execute StockMatCache.native(''swap cache names'');',
```

Native queries are used to simulate how its done using RDBMS and renaming tables, because Infinispan doesn't currently support renaming a cache. So the native queries will trigger the clearing of the "staging" cache, and the swapping of the cache aliases.

Direct Query Procedure

Additionally, the execution of native queries is done thru the support of direct query procedures. The procedure to be executed is called **native**.

Warning	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|Translators#Override Execution Properties] called **SupportsDirectQueryProcedure** to true.

Metadata Requirements

If you manually model the cache table in Teiid Designer, then you will need to add the property extension for defining the property "primary_table". The following is a DDL example:

```
SET NAMESPACE 'http://www.teiid.org/translator/object/2016' AS n0;

CREATE FOREIGN TABLE Trade (
         ....
        CONSTRAINT PK_TRADEID PRIMARY KEY(tradeId)
) OPTIONS (UPDATABLE TRUE);

CREATE FOREIGN TABLE ST_Trade (
        ....
) OPTIONS (NAMEINSOURCE 'Trade', UPDATABLE TRUE, "n0:primary_table" 'ObjectSchema.Trade');
```

JCA Resource Adapter

See JDG Library Mode Data Sources resource adapter for this translator. It can be configured to lookup the cache container via JNDI or created (i.e., ConfigurationFileName).

JDBC Translators

The JDBC translators bridge between SQL semantic and data type differences between Teiid and a target RDBMS. Teiid has a range of specific translators that target the most popular open source and proprietary databases.

Table of Contents

- Usage
- Execution Properties - shared by all JDBC Translators
- Importer Properties - shared by all JDBC Translators
  - Native Queries
  - Direct Query Procedure
- JCA Resource Adapter

Usage

Usage of a JDBC source is straight-forward. Using Teiid SQL, the source may be queried as if the tables and procedures were local to the Teiid system.

If you are using a relational data source, or a data source that has a JDBC driver, and you do not find a specific translator available for that data source type, then start with the JDBC ANSI Translator. The JDBC ANSI Translator should enable you to perform the SQL basics. If there specific data source capabilities that are not available, then consider using the Translator Development to create what you need. Or log a Teiid Jira with your requirements.

Execution Properties - shared by all JDBC Translators

Name	Description	Default
DatabaseTimeZone	The time zone of the database. Used when fetchings date, time, or timestamp values.	The system default time zone
DatabaseVersion	The specific database version. Used to further tune pushdown support.	The base supported version or derived from the DatabaseMetadata.getDatabaseProductVersion string. Automatic detection requires a Connection. If there are circumstances where you are getting an exception from capabilities being unavailable (most likely due to an issue obtaining a Connection), then set DatabaseVersion property. Use the JDBCExecutionFactory.usesDatabaseVersion() method to control whether your translator requires a connection to determine capabilities.
TrimStrings	true to trim trailing whitespace from fixed length character strings. Note that Teiid only has a string, or varchar, type that treats trailing whitespace as meaningful.	false

UseBindVariables	true to indicate that PreparedStatements should be used and that literal values in the source query should be replace with bind variables. If false only LOB values will trigger the use of PreparedStatements.	true
UseCommentsInSourceQuery	This will embed a leading comment with session/request id in the source SQL for informational purposes. Can be customized with the CommentFormat property.	false
CommentFormat	MessageFormat string to be used if UseCommentsInSourceQuery is enabled. Available properties: 0 - session id string, 1 - parent request id string, 2 - request part id string, 3 - execution count id string, 4 - user name string, 5 - vdb name string, 6 - vdb version integer, 7 - is transactional boolean	**/teiid sessionid:{0}, requestid:{1}.{2}/**
MaxPreparedInsertBatchSize	The max size of a prepared insert batch.	2048
StructRetrieval	Struct retrieval mode can be one of OBJECT - getObject value returned, COPY - returned as a `SerialStruct` , ARRAY - returned as an `Array`)	OBJECT
EnableDependentJoins	For sources that support temporary tables (DB2, Derby, H2, HSQL 2.0+, MySQL 5.0+, Oracle, PostgreSQL, SQLServer, Sybase) allow dependent join pushdown	false

Importer Properties - shared by all JDBC Translators

When specifying the importer property, it must be prefixed with "importer.". Example: importer.tableTypes

Name	Description	Default
catalog	See DatabaseMetaData.getTables [1]	null
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
useFullSchemaName	When false, directs the importer to drop the source catalog/schema from the Teiid object name, so that the Teiid fully qualified name will be in the form of <model name>.<table name> - Note: when false this may lead to objects with duplicate names when importing from multiple schemas, which results in an exception. This option does not affect the name in source property.	true
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]	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
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

quoteNameInSource	false will override the default and direct Teiid to create source queries using unquoted identifiers.	true
useProcedureSpecificName	true will allow the import of overloaded procedures (which will normally result in a duplicate procedure error) by using the unique procedur specific name as the Teiid name. This option will only work with JDBC 4.0 compatible drivers that report specific names.	false
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 useFullSchemaName is also true. false will not use the catalog name in either the name in source or the Teiid runtime name. Should be set to false for sources that do not fully support a catalog concept, but return a non-null catalog name in their metadata - such as HSQL.	true
useQualifiedName	true will use name qualification for both the Teiid name and name in source as dictated 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. Note: when false this may lead to objects with duplicate names when importing from multiple schemas, which results in an exception.	true
useAnyIndexCardinality	true will use the maximum cardinality returned from DatabaseMetaData.getIndexInfo. importKeys or importIndexes needs to be enabled for this setting to have an effect. This allows for better stats gathering from sources that don’t support returning a statistical index.	false
importStatistics	true will use database dependent logic to determine the cardinality if none is determined. Not yet supported by all database types - currently only supported by Oracle and MySQL.	false
importRowIdAsBinary	true will import RowId columns as varbinary values.	false

[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 schemaPattern and tableTypes.

Example importer settings to only import tables and views from my-schema. See also VDB Guide

```
<model ...

  <property name="importer.tableTypes" value="TABLE,VIEW"/>
  <property name="importer.schemaPattern" value="my-schema"/>
  ...
</model>
```

Native Queries

Physical tables, functions, and procedures may optionally have native queries associated with them. No validation of the native query is performed, it is simply used in a straight-forward manner to generate the source SQL. For a physical table setting the teiid_rel:native-query extension metadata will execute the native query as an inline view in the source query. This feature should only be used against sources that support inline views. The native query is used as is and is not treated as a parameterized string. For example on a physical table y with nameInSource="x" and teiid_rel:native-query="select c from g", the Teiid source query"SELECT c FROM y" would generate the SQL query "SELECT c FROM (select c from g) as x". Note that the column names in the native query must match the nameInSource of the physical table columns for the resulting SQL to be valid.

For physical procedures you may also set the teiid_rel:native-query extension metadata to a desired query string with the added ability to positionally reference IN parameters - see Parameterizable Native Queries. The teiid_rel:non-prepared extension metadata property may be set to false to turn off parameter binding. Note this option should be used with caution as inbound may allow for SQL injection attacks if not properly validated. The native query does not need to call a stored procedure. Any SQL that returns a result set positionally matching the result set expected by the physical stored procedure metadata will work. For example on a stored procedure x with teiid_rel:native-query="select c from g where c1 = $1 and c2 = `$$1'", the Teiid source query "CALL x(?)" would generate the SQL query "select c from g where c1 = ? and c2 = `$1'". Note that ? in this example will be replaced with the actual value bound to parameter 1.

Direct Query Procedure

This feature is turned off by default because of the security risk this exposes to execute any command against the source. To enable this feature, override the execution property called _SupportsDirectQueryProcedure to true.

By default the name of the procedure that executes the queries directly is **native**. Override the execution property _DirectQueryProcedureName to change it to another name.

The JDBC translator provides a procedure to execute any ad-hoc SQL query directly against the source without Teiid parsing or resolving. Since the metadata of this procedure's results are not known to Teiid, they are returned as an object array. ARRAYTABLE can be used construct tabular output for consumption by client applications.

Select Example

```
SELECT x.* FROM (call jdbc_source.native('select * from g1')) w,
 ARRAYTABLE(w.tuple COLUMNS "e1" integer , "e2" string) AS x
```

Insert Example

```
SELECT x.* FROM (call jdbc_source.native('insert into g1 (e1,e2) values (?, ?)', 112, 'foo')) w,
 ARRAYTABLE(w.tuple COLUMNS "update_count" integer) AS x
```

Update Example

```
SELECT x.* FROM (call jdbc_source.native('update g1 set e2=? where e1 = ?','blah', 112)) w,
 ARRAYTABLE(w.tuple COLUMNS "update_count" integer) AS x
```

Delete Example

```
SELECT x.* FROM (call jdbc_source.native('delete from g1 where e1 = ?', 112)) w,
 ARRAYTABLE(w.tuple COLUMNS "update_count" integer) AS x
```

JCA Resource Adapter

The resource adapter for this translator provided through data source in WildFly, See to Admin Guide section WildFly Data Sources for configuration.

Actian Vector Translator (actian-vector)

Also see common JDBC Translator Information

The Actian Vector Translator, known by the type name ***actian-vector***, is for use Actian Vector in Hadoop.

Download the JDBC driver at http://esd.actian.com/platform. Note the port number in connection URL is "AH7" which maps to 16967.

Apache HBase Translator (phoenix)

Also see common JDBC Translator Information

The Apache Phoenix Translator, known by the type name ***phoenix***, exposes querying functionality to HBase Tables. Apache Phoenix is a JDBC SQL interface for HBase - see Phoenix Data Sources that is required for this translator as it pushes down commands into Phoenix SQL.

The translator is also known by the deprecated name ***hbase***. The name change reflects that the translator is specific to phoenix and there may be other translators introduced in the furture that also connect to HBase.

The DatabaseTimezone property should not be used with this translator.

The HBase Translator doesn’t support Joins. Phoenix uses the HBase Table Row ID as the Primary Key, which map to . This Translator is developed with Phoenix 4.3+ for HBase 0.98.1+.

Note	The translator implements INSERT/UPDATE through the Phoenix UPSERT operation. This means you can see different behavior than with standard INSERT/UPDATE - such as repeated inserts will not throw a duplicate key exception, but will instead update the row in question.

Note	Due to Phoenix driver limitations the importer will not look for unique constraints and defaults to not importing foreign keys.

Note	The translator supports offset and other features starting with Phoenix 4.8. The Phoenix driver hard codes the server version in PhoenixDatabaseMetaData, and does not otherwise provide a way to detect the server version at runtime. If a newer driver is used with an older server, please set the database version translator property manually.

Warning	The Phoenix driver does not have robust handling of time values. If your time values are normalized to use a date component of 1970-01-01, then the default handling will work correctly. If not, then the time column should be modeled as timestamp instead.

Cloudera Impala Translator (impala)

Also see common JDBC Translator Information

The Coudera Impala Translator, known by the type name ***impala***, is for use with Cloudera Impala 1.2.1 or later.

Impala has limited support for data types. It is does not have native support for time/date/xml or LOBs. These limitations are reflected in the translator capabilities. A Teiid view can use these types, however the transformation would need to specify the necessary conversions. Note that in those situations, the evaluations will be done in the Teiid engine.

The DatabaseTimeZone translator property should not be used.

Impala only supports EQUI join, so using any other joins types on its source tables will result in inefficient queries.

To write criteria based on partitioned columns, modeled them on source table, but do not include them in selection columns.

Note	Impala Hive importer does not have concept of catalog or source schema, nor does it import keys, procedures, indexes, etc.

Impala specific importer properties:

useDatabaseMetaData- Set to true to use the normal import logic with the option to import index information disabled. Defaults to false.

Note	If useDatabaseMetaData is false the typical JDBC DatabaseMetaData calls are not used so not all of the common JDBC importer properties are applicable to Impala. You may still use excludeTables regardless.

Note	Some versions of Impala requires the use of a LIMIT when performing an ORDER BY. If no default is configured in Impala, then an exception can occur when a Teiid query with an ORDER BY but no LIMIT is issued. You should set an Impala wide default, or configure the connection pool to use a new connection sql string to issue a SET DEFAULT_ORDER_BY_LIMIT statement. See the Cloudera docs for more on limit options - such as controlling what happens when the limit is exceeded.

Note	The Impala JDBC driver seems to have issues with PreparedStatements and statement parsing in general that may require disabling useBindVariables - see link:https://issues.jboss.org/browse/TEIID-4610

DB2 Translator (db2)

Also see common JDBC Translator Information

The DB2 Translator, known by the type name ***db2***, if for use with DB2 8 or later and DB2 for i 5.4 or later.

DB2 specific execution properties:

- *DB2ForI*- indicates that the the DB2 instance is DB2 for i. Defaults to false.
- *supportsCommonTableExpressions*- indicates that the DB2 instance supports Common Table Expressions. Defaults to true. Some older versions, or instances running in a conversion mode, of DB2 lack full common table expression support and may need support disabled.

Derby Translator (derby)

Also see common JDBC Translator Information

The Derby Translator, known by the type name ***derby***, is for use with Derby 10.1 or later.

Greenplum Translator (greenplum)

Also see common JDBC Translator Information

The Greenplum Translator, known by the type name ***greenplum***, is for use with the Greenplum database. This translator is an extension of the PostgreSQL Translator and inherits its options.

H2 Translator (h2)

Also see common JDBC Translator Information

The H2 Translator, known by the type name ***h2***, is for use with H2 version 1.1 or later.

Hive Translator (hive)

Also see common JDBC Translator Information

The Hive Translator, known by the type name ***hive***, is for use with Hive v.10 and SparkSQL v1.0 and later.

Capabilities

Hive has limited support for data types. It does not have native support for time/xml or LOBs. These limitations are reflected in the translator capabilities. A Teiid view can use these types, however the transformation would need to specify the necessary conversions. Note that in those situations, the evaluations will be done in Teiid engine.

The DatabaseTimeZone translator property should not be used.

Hive only supports EQUI join, so using any other joins types on its source tables will result in inefficient queries.

To write criteria based on partitioned columns, modeled them on source table, but do not include them in selection columns.

Note	The Hive importer does not have concept of catalog or source schema, nor does it import keys, procedures, indexes, etc.

Import Properties

- *trimColumnNames*- For Hive 0.11.0 and later the the DESCRIBE command metadata is inappropriately returned with padding, set to true to strip trim white space from column names. Defaults to false.
- *useDatabaseMetaData*- For Hive 0.13.0 and later the normal JDBC DatabaseMetaData facilities are sufficient to perform an import. Set to true to use the normal import logic with the option to import index information disabled. Defaults to false. When true, trimColumnNames has no effect.

Note	If false the typical JDBC DatabaseMetaData calls are not used so not all of the common JDBC importer properties are applicable to Hive. You may still use excludeTables regardless.

"Database Name"

When the database name used in the Hive is different than "default", the metadata retrieval and execution of queries does not work as expected in Teiid, as Hive JDBC driver seems to be implicitly connecting (tested with < 0.12) to "default" database, thus ignoring the database name mentioned on connection URL. This can workaround in the Teiid in WildFly environment by setting the following in data source configuration.

```
    <new-connection-sql>use {database-name}</new-connection-sql>
```

This is fixed in > 0.13 version Hive Driver. See https://issues.apache.org/jira/browse/HIVE-4256

HSQL Translator (hsql)

Also see common JDBC Translator Information

The HSQL Translator, known by the type name ***hsql***, is for use with HSQLDB 1.7 or later.

Informix Translator (informix)

Also see common JDBC Translator Information

The Informix Translator, known by the type name ***informix***, is for use with any Informix version.

Known Issues

TEIID-3808 - The Informix driver handling of timezone information is inconsistent - even if the databaseTimezone translator property is set. Consider ensuring that the Informix server and the application server are in the same timezone.

Ingres Translators (ingres / ingres93)

Also see common JDBC Translator Information

The Ingres translation is supported by 2 translators.

ingres

The Ingres Translator, known by the type name ***ingres***, is for use with Ingres 2006 or later.

ingres93

The Ingres93 Translator, known by the type name ***ingres93***, is for use with Ingres 9.3 or later.

Intersystems Cache Translator (intersystems-cache)

Also see common JDBC Translator Information

The Intersystem Cache Translator, known by the type name ***intersystems-cache***, is for use with Intersystems Cache Object database (only relational aspect of it).

JDBC ANSI Translator (jdbc-ansi)

Also see common JDBC Translator Information

The JDBC ANSI translator, known by the type name ***jdbc-ansi***, declares support for most SQL constructs supported by Teiid, except for row limit/offset and EXCEPT/INTERSECT. Translates source SQL into ANSI compliant syntax. This translator should be used when another more specific type is not available. If source exceptions arise from unsupported SQL, then consider using the JDBC Simple Translator to further restrict capabilities, or create a Custom Translator / create an enhancement request.

JDBC Simple Translator (jdbc-simple)

Also see common JDBC Translator Information

The JDBC Simple translator, known by the type name ***jdbc-simple***, is the same as jdbc-ansi, except disables support for nearly all pushdown constructs for maximum compatibility.

MetaMatrix Translator (metamatrix)

Also see common JDBC Translator Information

The MetaMatrix Translator, known by the type name ***metamatrix***, is for use with MetaMatrix 5.5.0 or later.

Microsoft Access Translators

Also see common JDBC Translator Information

access

The Microsoft Access Translator known by the type name ***access*** is for use with Microsoft Access 2003 or later via the JDBC-ODBC bridge.

If you are using the default native metadata import or the Teiid connection importer the importer defaults to importKeys=false and excludeTables=.**[.]MSys.** to avoid issues with the metadata provided by the JDBC ODBC bridge. You may need to adjust these values if you use a different JDBC driver.

ucanaccess

The Microsoft Access Translator known by the type name ***ucanaccess*** is for use with Microsoft Access 2003 or later via the for the UCanAccess driver.

Microsoft SQL Server Translator (sqlserver)

Also see common JDBC Translator Information

The Microsoft SQL Server Translator, known by the type name ***sqlserver***, is for use with SQL Server 2000 or later. A SQL Server JDBC driver version 2.0 or later (or compatible e.g. JTDS 1.2 or later) should be used. The SQL Server DatabaseVersion property may be set to 2000, 2005, 2008, or 2012, but otherwise expects a standard version number - e.g. "10.0".

Sequence Support

With Teiid 8.5+ sequence operations may be modeled as source functions.

Example:Sequence Native Query

```
CREATE FOREIGN FUNCTION seq_nextval () returns integer OPTIONS ("teiid_rel:native-query" 'NEXT VALUE FOR seq');
```

Execution Properties

SQL Server specific execution properties:

- *JtdsDriver*- indicates that the open source JTDS driver is being used. Defaults to false.

ModeShape Translator (modeshape)

Also see common JDBC Translator Information

The ModeShape Translator, known by the type name ***modeshape***, is for use with Modeshape 2.2.1 or later.

Usage

The PATH, NAME, LOCALNODENAME, DEPTH, and SCORE functions should be accessed as pseudo-columns, e.g. "nt:base"."jcr:path".

Teiid UFDs (prefixed by JCR_) are available for CONTIANS, ISCHILDNODE, ISDESCENDENT, ISSAMENODE, REFERENCE - see the JCRFunctions.xmi. If a selector name is needed in a JCR function, you should use the pseudo-column "jcr:path", e.g. JCR_ISCHILDNODE(foo.jcr_path, 'x/y') would become ISCHILDNODE(foo, `x/y') in the ModeShape query.

An additional pseudo-column "mode:properties" should be imported by setting the ModeShape JDBC connection property teiidsupport=true. The column "mode:properties" should be used by the JCR_REFERENCE and other functions that expect a .* selector name, e.g. JCR_REFERENCE(nt_base.jcr_properties) would become REFERENCE("nt:base".*) in the ModeShape query.

MySQL Translator (mysql/mysql5)

Also see common JDBC Translator Information

MySQL/MariaDB translation is supported by 2 translators.

mysql

The Mysql translator, known by the type name ***mysql***, is for use with MySQL version 4.x.

mysql5

The Mysql5 translator, known by the type name ***mysql5***, is for use with MySQL version 5 or later.

Also supports compatible MySQL derivatives including MariaDB.

Usage

The MySQL Translators expect the database or session to be using ANSI mode. If the database is not using ANSI mode, an initialization query should be used on the pool to set ANSI mode:

```
set SESSION sql_mode = 'ANSI'
```

If you may deal with null timestamp values, then set the connection property zeroDateTimeBehavior=convertToNull. Otherwise you'll get conversion errors in Teiid that `0000-00-00 00:00:00' cannot be converted to a timestamp.

Warning	If retrieving large result sets, you should consider setting the connection property useCursorFetch=true, otherwise MySQL will fully fetch result sets into memory on the Teiid instance.

Note	MySQL reports TINYINT(1) columns as a JDBC BIT type - however the value range is not actually restricted and may cause issues if for example you are relying on -1 being recognized as a true value. If not using the native importer, you should change affected source BOOLEAN columns to have a native type of "TINYINT(1)" rather than BIT so that the translator can appropriately handle the boolean conversion.

Netezza Translator (netezza)

Also see common JDBC Translator Information

The Netezza Translator, known by the type name ***netezza***, is for use with any Netezza version.

Usage

The current Netezza vendor supplied JDBC driver performs poorly with single transactional updates. As is generally the case when possible use batched updates.

Execution Properties

Netezza specific execution properties:

- *SqlExtensionsInstalled*- indicates that SQL Extensions, including support for REGEXP_LIKE, are installed. All other REGEXP functions are then available as pushdown functions. Defaults to false.

Oracle Translator (oracle)

Also see common JDBC Translator Information

The Oracle Translator, known by the type name ***oracle***, is for use with Oracle 9i or later.

Importer Properties

- *useGeometryType*- Use the Teiid Geomety type when importing columns with a source type of SDO_GEOMETRY. Defaults to false.
- *useIntegralTypes*- Use integral types rather than decimal when the scale is 0. Defaults to false.

Execution Properties

- *OracleSuppliedDriver*- indicates that the Oracle supplied driver (typically prefixed by ojdbc) is being used. Defaults to true. Set to false when using DataDirect or other Oracle JDBC drivers.

Oracle Specific Metadata

Sequences

Sequences may be used with the Oracle translator. A sequence may be modeled as a table with a name in source of DUAL and columns with the name in source set to <sequence name>.[nextval|currval]

Teiid 8.4 and Prior Oracle Sequence DDL

```
CREATE FOREIGN TABLE seq (nextval integer OPTIONS (NAMEINSOURCE 'seq.nextval'), currval integer options (NAMEIN
SOURCE 'seq.currval') ) OPTIONS (NAMEINSOURCE 'DUAL')
```

With Teiid 8.5 it's no longer necessary to rely on a table representation and Oracle specific handling for sequences. See DDL Metadata for representing currval and nextval as source functions.

8.5 Example:Sequence Native Query

```
CREATE FOREIGN FUNCTION seq_nextval () returns integer OPTIONS ("teiid_rel:native-query" 'seq.nextval');
```

You can also use a sequence as the default value for insert columns by setting the column to autoincrement and the name in source to `<element name>:SEQUENCE=<sequence name>.<sequence value>`.

Rownum

A rownum column can also added to any Oracle physical table to support the rownum pseudo-column. A rownum column should have a name in source of `rownum`. These rownum columns do not have the same semantics as the Oracle rownum construct so care must be taken in their usage.

Out Parameter Result Set

Out parameters for procedures may also be used to return a result set, if this is not represented correctly by the automatic import you need to manually create a result set and represent the output parameter with native type "REF CURSOR".

DDL for out parameter result set

```
create foreign procedure proc (in x integer, out y object options (native_type 'REF CURSOR'))
returns table (a integer, b string)
```

Geo Spatial function support

Oracle translator supports geo spatial functions. The supported functions are:

Relate = sdo_relate

```
CREATE FOREIGN FUNCTION sdo_relate (arg1 string,  arg2 string,  arg3 string) RETURNS string;
CREATE FOREIGN FUNCTION sdo_relate (arg1 Object,  arg2 Object,  arg3 string) RETURNS string;
CREATE FOREIGN FUNCTION sdo_relate (arg1 string,  arg2 Object,  arg3 string) RETURNS string;
CREATE FOREIGN FUNCTION sdo_relate (arg1 Object,  arg2 string,  arg3 string) RETURNS string;
```

Nearest_Neighbor = sdo_nn

```
CREATE FOREIGN FUNCTION sdo_nn (arg1 string,  arg2 Object,  arg3 string,  arg4 integer) RETURNS string;
CREATE FOREIGN FUNCTION sdo_nn (arg1 Object,  arg2 Object,  arg3 string,  arg4 integer) RETURNS string;
CREATE FOREIGN FUNCTION sdo_nn (arg1 Object,  arg2 string,  arg3 string,  arg4 integer) RETURNS string;
```

Within_Distance = sdo_within_distance

```
CREATE FOREIGN FUNCTION sdo_within_distance (arg1 Object,  arg2 Object,  arg3 string) RETURNS string;
CREATE FOREIGN FUNCTION sdo_within_distance (arg1 string,  arg2 Object,  arg3 string) RETURNS string;
CREATE FOREIGN FUNCTION sdo_within_distance (arg1 Object,  arg2 string,  arg3 string) RETURNS string;
```

Nearest_Neigher_Distance = sdo_nn_distance

```
CREATE FOREIGN FUNCTION sdo_nn_distance (arg integer) RETURNS integer;
```

Filter = sdo_filter

```
CREATE FOREIGN FUNCTION sdo_filter (arg1 Object,  arg2 string,  arg3 string) RETURNS string;
CREATE FOREIGN FUNCTION sdo_filter (arg1 Object,  arg2 Object,  arg3 string) RETURNS string;
CREATE FOREIGN FUNCTION sdo_filter (arg1 string,  arg2 object,  arg3 string) RETURNS string;
```

OSISoft PI Translator (osisoft-pi)

Also see common JDBC Translator Information

The OSISoft Translator, known by the type name ***osisoft-pi***, is for use with OSIsoft PI OLEDB Enterprise. This translator uses the JDBC driver provided by the OSISoft. The driver is not provided with Teiid install, this needs be downloaded from OSISoft and installed correctly on Teiid server according to OSISoft documentation *PI-JDBC-2016-Administrator-Guide.pdf* or latest document.

Install on Linux

Make sure you have OpenSSL libraries installed, and you have following "export" added correctly in your shell environment variables. Otherwise you can also add in *<WildFly>/bin/standalone.sh* file or *<WildFly>/bin/domain.sh* file.

```
export PI_RDSA_LIB=/<path>/pipc/jdbc/lib/libRdsaWrapper-1.5b.so
export PI_RDSA_LIB64=/<path>/pipc/jdbc/lib/libRdsaWrapper64-1.5b.so
```

Please also note to execute from Linux, you also need install 'gSoap' library, as PI JDBC driver uses SOAP over HTTPS to communicate with PI server.

Install on Windows

Follow the installation program provided by OSISoft for installing the JDBC drivers. Make sure you have the following environment variables configured.

```
PI_RDSA_LIB     C:\Program Files (x86)\PIPC\JDBC\RDSAWrapper.dll
PI_RDSA_LIB64   C:\Program Files\PIPC\JDBC\RDSAWrapper64.dll
```

Installing the JDBC driver for Teiid (same for both Linux and Windows)

Then copy the module directory from *<WildFly>/teiid/datasources/osisoft-pi/modules directory into _<WilfFly>/modules* directory. Then find the "PIJDBCDriver.jar" file from the installation directory, and copy it to _<WildFly>/module/system/layers/dv/com/osisoft/main" directory. Then add the driver definition to the standalone.xml file by editing the file and adding something similar to below

```
    <drivers>
       <driver name="osisoft-pi" module="com.osisoft">
           <driver-class>com.osisoft.jdbc.Driver</driver-class>
       </driver>
    </drivers>
```

That completes the configuration of the PI driver in the Teiid. We still have not created a connection to the PI server. you can start the server now.

Creating a Data Source to PI

You can execute following similar CLI script to create a datasource

```
/subsystem=datasources/data-source=pi-ds:add(jndi-name=java:/pi-ds,  driver-
name=osisoft-pi, connection-url=jdbc:pioledbent://<DAC Server>/Data Source=<AF
Server>; Integrated Security=SSPI,user-name=user, password=mypass)
/subsystem=datasources/data-source=pi-ds:enable
```

this will create following XML in standalone.xml or domain.xml (you can also directly edit these files and add manually)

```
<datasource jndi-name="java:/pi-ds" pool-name="pi-ds">
    <connection-url>jdbc:pioledbent://<DAC Server>/Data Source=<AF Server>;
Integrated Security=SSPI</connection-url>
    <driver>osisoft-pi</driver>
    <pool>
        <prefill>false</prefill>
        <use-strict-min>false</use-strict-min>
        <flush-strategy>FailingConnectionOnly</flush-strategy>
    </pool>
    <security>
        <user-name>user</user-name>
        <password>mypass</password>
    </security>
</datasource>
```

Now you have fully configured the Teiid with PI database connection. You can create VDB that can use this connection to issue the queries.

Usage

You can develop a VDB like follows to fetch metadata from PI and give you access to executing queries against PI.

pi-vdb.xml

```
<vdb name="pi" version="1">
    <model name="AF">
        <property name="importer.importProcedures" value="true"/>
        <source connection-jndi-name="java:/pi-ds" name="pi-connector" translator-
name="osisoft-pi"/>
    </model>
</vdb>
```

Deploy this file into Teiid using CLI or using management console

```
deploy pi-vdb.xml
```

Once the metadata is loaded and VDB is active you can use Teiid JDBC/ODBC driver or OData to connect to the VDB and issue queries.

PI Translator Capabilities

PI translator is extension of *jdbc-ansi* translator, so all the SQL ANSI queries are supported. PI translator also supports LATERAL join with Table Valued Functions (TVF). An example Teiid query looks like

{code} 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\') {code}

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.

PostgreSQL Translator (postgresql)

Also see common JDBC Translator Information

The PostgreSQL Translator, known by the type name ***postgresql***, is for use with 8.0 or later clients and 7.1 or later server.

Execution Properties

PostgreSQL specific execution properties:

- *PostGisVersion*- indicate the PostGIS version in use. Defaults to 0 meaning PostGIS is not installed. Will be set automatically if the database version is not set.
- *ProjSupported*- boolean indicating if Proj is support for PostGis. Will be set automatically if the database version is not set.

Note	Some driver versions of PostgreSQL will not associate columns to "INDEX" type tables and fail during import if the importer.tableType property is left as the default.

PrestoDB Translator (prestodb)

Also see common JDBC Translator Information

The PrestoDB translator, known by the type name ***prestodb***, exposes querying functionality to PrestoDB Data Sources. In data integration respect, PrestoDB has very similar capabilities of Teiid, however it goes beyond in terms of distributed query execution with multiple worker nodes. Teiid's execution model is limited to single execution node and focuses more on pushing the query down to sources. Currently Teiid has much more complete query support and many enterprise features.

Capabilities

The PrestoDB translator supports only SELECT statements with a restrictive set of capabilities. This translator is developed with 0.85 version of PrestoDB and capabilities are designed for this version. With new versions of PrestoDB Teiid will adjust the capabilities of this translator. Since PrestoDB exposes a relational model, the usage of this is no different than any RDBMS source like Oracle, DB2 etc. For configuring the PrestoDB consult the PrestoDB documentation.

Tip	PrestoDB not support multiple columns in the ORDER BY with in JOIN situations well, the translator property `supportsOrderBy` can use to disable Order by in some specific situations.

Tip	Some versions of PrestoDB not support null as valid values in subquery well, to check the PrestoDB whether support null as valid values in subquery if you hit releated error.

Tip	PrestoDB not support transaction, define a no-tx-datasource is recommend.

Note	Every catalogs in PrestoDB has a `information_schema` schema by default, If configure multiple catalogs, it should be consider to use import options to filter the schemas/tables, to avoid `Duplicate Table` error cause VDB deploy failed. For instance, set `catalog` to a specific catalog name to match the catalog name as it is stored in the prestodb, set `schemaPattern` to a regular expression to filter schemas by matching result, set `excludeTables` to a regular expression to filter tables by matching result.

Note	PrestoDB JDBC driver uses Joda-Time library to work with time/date/timestamp. If you need to customize server's time zone (e.g. setting -Duser.timezone via JAVA_OPTS), you cannot use `GMT/…` ID as Joda-Time does not regonize it. However, you can use equivalent `Etc/…` ID. For more details see Joda-Time timezones.

Redshift Translator (redshift)

Also see common JDBC Translator Information

The Redshift Translator, known by the type name ***redshift***, is for use with the Redshift database. This translator is an extension of the PostgreSQL Translator and inherits its options.

SAP Hana Translator (hana)

Also see common JDBC Translator Information

The SAP Hana Translator, known by the name of ***hana***, is for use with SAP Hana.

Known Issues

TEIID-3805 - The pushdown of the substring function is inconsistent with the Teiid substring function when the from index exceeds the length of the string. SAP Hana will return an empty string, while Teiid produces a null value.

SybaseIQ Translator (sybaseiq)

Also see common JDBC Translator Information

The SybaseIQ Translator, known by the type name ***sybaseiq***, is for use with Sybase IQ version 15.1 or later.

Sybase Translator (sybase)

The Sybase Translator, known by the type name ***sybase***, is for use with Sybase version 12.5 or later.

If using the the default native import and no import properties are specified (not recommended, see import properties below), then exceptions can be thrown retrieving system table information. You should specify a schemaPattern or use excludeTables to exclude system tables if this occurs.

If the name in source metadata contains quoted identifiers (such as required by reserved words or words containing characters that would not otherwise be allowed) and you are using a jconnect Sybase driver, you must first configure the connection pool to enable **quoted_identifier**:

Driver URL with SQLINITSTRING

```
jdbc:sybase:Tds:host.at.some.domain:5000/db_name?SQLINITSTRING=set quoted_identifier on
```

If you are a jconnect Sybase driver and will target the source for dependent joins, you should allow the translator to send more values by setting the JCONNECT_VERSION. Otherwise you will get exceptions with statements that have more than 481 bind values:

Driver URL with JCONNECT_VERSION

```
jdbc:sybase:Tds:host.at.some.domain:5000/db_name?SQLINITSTRING=set quoted_identifier on&JCONNECT_VERSION=6
```

Sybase specific execution properties:

- *JtdsDriver*- indicates that the open source JTDS driver is being used. Defaults to false.

Teiid Translator (teiid)

Also see common JDBC Translator Information

The Teiid Translator, known by the type name ***teiid***, is for use with Teiid 6.0 or later.

Teradata Translator (teradata)

Also see common JDBC Translator Information

The Teradata Translator, known by the type name ***teradata***, is for use with Teradata V2R5.1 or later.

With Teradata driver version 15 date, time, and timestamp values by default will be adjusted for the Teiid server timezone. To remove this adjustment, set the translator DatabaseTimezone property to GMT or whatever the Teradata server defaults to.

Vertica Translator (vertica)

Also see common JDBC Translator Information

The Vertica Translator, known by the type name ***vertica***, is for use with Vertica 6 or later.

JPA Translator

The JPA translator, known by the type name *jpa2*, can reverse a JPA object model into a relational model, which can then be integrated with other relational or non-relational sources. For information on JPA persistence in a WildFly, see JPA Reference Guide.

Properties

The JPA Translator currently has no import or execution properties.

Native Queries

JPA source procedures may be created using the teiid_rel:native-query extension - see Parameterizable Native Queries. The procedure will invoke the native-query similar to an native procedure call with the benefits that the query is predetermined and that result column types are known, rather than requiring the use of ARRAYTABLE or similar functionality. See the query syntax below.

Direct Query Procedure

Note	This feature is turned off by default because of the security risk this exposes to execute any command against the source. To enable this feature, override the execution property called _SupportsDirectQueryProcedure to true.

Tip	By default the name of the procedure that executes the queries directly is **native**. Override the execution property _DirectQueryProcedureName to change it to another name.

The JPA translator provides a procedure to execute any ad-hoc JPA-QL query directly against the source without Teiid parsing or resolving. Since the metadata of this procedure's results are not known to Teiid, they are returned as object array. User can use ARRAYTABLE can be used construct tabular output for consumption by client applications. Teiid exposes this procedure with a simple query structure as below

Select

Select Example

```
SELECT x.* FROM (call jpa_source.native('search;FROM Account')) w,
 ARRAYTABLE(w.tuple COLUMNS "id" string , "type" string, "name" String) AS x
```

from the above code, the "search" keyword followed by a query statement - see Parameterizable Native Queries to substitute parameter values.

Delete

Delete Example

```
SELECT x.* FROM (call jpa_source.native('delete;<jpa-ql>')) w,
 ARRAYTABLE(w.tuple COLUMNS "updatecount" integer) AS x
```

form the above code, the "delete" keyword followed by JPA-QL for delete operation.

Update

Create Example

```
SELECT x.* FROM
 (call jpa_source.native('update;<jpa-ql>')) w,
 ARRAYTABLE(w.tuple COLUMNS "update_count" integer) AS x
```

form the above code, the "update" keyword must be followed by JPA-QL for the update statement.

Create

Update Example

```
SELECT x.* FROM
 (call jpa_source.native('create;', <entity>)) w,
 ARRAYTABLE(w.tuple COLUMNS "update_count" integer) AS x
```

Create operation needs to send "create" word as marker and send the entity as a the first parameter.

LDAP Translator

The LDAP translator is implemented by the org.teiid.translator.ldap.LDAPExecutionFactory class and known by the translator type name ldap. The LDAP translator exposes an LDAP directory tree relationally with pushdown support for filtering via criteria. This is typically coupled with the LDAP resource adapter.

Note	The resource adapter for this translator is provided by configuring the ldap data source in the JBoss EAP instance.

Execution Properties

Name	Description	Default
SearchDerfaultBaseDN	Default Base DN for LDAP Searches	null
SearchDefaultScope	Default Scope for LDAP Searches. Can be one of SUBTREE_SCOPE, OBJECT_SCOPE, ONELEVEL_SCOPE.	ONELEVEL_SCOPE
RestrictToObjectClass	Restrict Searches to objectClass named in the Name field for a table	false
UsePagination	Use a PagedResultsControl to page through large results. This is not supported by all directory servers.	false
ExceptionOnSizeLimitExceeded	Set to true to throw an exception when a SizeLimitExceededException is received and a LIMIT is not properly enforced.	false

There are no import settings for the ldap translator; it also does not provide metadata.

Metadata Options

SEARCHABLE 'equality_only'

For openldap, apacheds, and other ldap servers dn attributes have search restrictions, such that only equality predicates are supported. Use SEARCHABLE `equality_only` to indicates that only equality predicates should be pushed down. Any other predicate would need evaluated in the engine. For example

```
col string OPTIONS (SEARCHABLE 'equality_only', ...)
```

teiid_ldap:rdn_type

Used on a column with a dn value to indicate the rdn to extract. If the entry suffix does not match this rdn type, then no row will be produced. For example

```
col string OPTIONS ("teiid_ldap:rdn_type" 'cn', ...)
```

teiid_ldap:dn_prefix

Used on a column if rdn_type is specified to indicates that the values should match this prefix, no row will be produced for a non-matching entry. For example

```
col string OPTIONS ("teiid_ldap:rdn_type" 'cn', "teiid_ldap:dn_prefix" 'ou=groups,dc=example,dc=com', ...)
```

Multivalued Attribute Support

If one of the methods below is not used and the attribute is mapped to a non-array type, then any value may be returned on a read operation. Also insert/update/delete support will not be multi-value aware.

Concatenation

String columns with a default value of "multivalued-concat" will concatenate all attribute values together in alphabetical order using a ? delimiter. Insert/update will function as expected if all applicable values are supplied in the concatenated format.

Array support

Multiple attribute values may also supported as an array type. The array type mapping also allows for insert/update operations.

For example here is ddl with objectClass and uniqueMember as arrays:

```
create foreign table ldap_groups (objectClass string[], DN string, name string options (nameinsource 'cn'), uni
queMember string[]) options (nameinsource 'ou=groups,dc=teiid,dc=org', updatable true)
```

The array values can be retrieved with a SELECT. An example insert with array values could look like:

```
insert into ldap_groups (objectClass, DN, name, uniqueMember) values (('top', 'groupOfUniqueNames'), 'cn=a,ou=g
roups,dc=teiid,dc=org', 'a', ('cn=Sam Smith,ou=people,dc=teiid,dc=org',))
```

Unwrap

When a multivalued attribute represents an association between entities, it’s possible to use extension metadata properties to represent it as a 1-to-many or many-to-many relationship.

Example many-to-many DDL:

```
CREATE foreign table users (username string primary key options (nameinsource 'cn'), surname string options (na
meinsource 'sn'), ...) options (nameinsource 'ou=users,dc=example,dc=com');

CREATE foreign table groups (groupname string primary key options (nameinsource 'cn'), description string, ...)
 options (nameinsource 'ou=groups,dc=example,dc=com');

CREATE foreign table membership (username string options (nameinsource 'cn'), groupname options (nameinsource '
memberOf', SEARCHABLE 'equality_only', "teiid_rel:partial_filter" true, "teiid_ldap:unwrap" true, "teiid_ldap:d
n_prefix" 'ou=groups,dc=example,dc=com', "teiid_ldap:rdn_type" 'cn'), foreign key (username) references users (
username), foreign key (groupname) references groups (groupname)) options (nameinsource 'ou=users,dc=example,dc
=com');
```

The result from "select * from membership" will then produce 1 row for each memberOf and the key value will be based upon the cn rdn value rather than the full dn. Also queries that join between users and membership will be pushed as a single query.

If the unwrap attribute is missing or there are no values, then a single row with a null value will be produced.

Native Queries

LDAP procedures may optionally have native queries associated with them - see Parameterizable Native Queries. The operation prefix (select;, insert;, update;, delete; - see below for more) must be present in the native-query, butit will not be issued as part of the query to the

Example DDL for an LDAP native procedure

```
CREATE FOREIGN PROCEDURE proc (arg1 integer, arg2 string) OPTIONS ("teiid_rel:native-query" 'search;context-nam
e=corporate;filter=(&(objectCategory=person)(objectClass=user)(!cn=$2));count-limit=5;timeout=$1;search-scope=O
NELEVEL_SCOPE;attributes=uid,cn') returns (col1 string, col2 string);
```

Parameter values will have reserved characters escaped, but are otherwise directly substituted into the query.

Direct Query Procedure

Note	This feature is turned off by default because of the security risk this exposes to execute any command against the source. To enable this feature, override the execution property called _SupportsDirectQueryProcedure to true.

Tip	By default the name of the procedure that executes the queries directly is **native**. Override the execution property _DirectQueryProcedureName to change it to another name.

The LDAP translator provides a procedure to execute any ad-hoc LDAP query directly against the source without Teiid parsing or resolving. Since the metadata of this procedure's results are not known to Teiid, they are returned as an object array. ARRAYTABLE can be used construct tabular output for consumption by client applications.

Search

Search Example

```
SELECT x.* FROM (call pm1.native('search;context-name=corporate;filter=(objectClass=*);count-limit=5;timeout=6;
search-scope=ONELEVEL_SCOPE;attributes=uid,cn')) w,
 ARRAYTABLE(w.tuple COLUMNS "uid" string , "cn" string) AS x
```

from the above code, the "**search**" keyword followed by below properties. Each property must be delimited by semi-colon (;) If a property contains a semi-colon (;), it should be escaped by another semi-colon - see alsoParameterizable Native Queries and the native-query procedure example above.

Name	Description	Required
context-name	LDAP Context name	Yes
filter	query to filter the records in the context	No
count-limit	limit the number of results. same as using LIMIT	No
timeout	Time out the query if not finished in given milliseconds	No
search-scope	LDAP search scope, one of SUBTREE_SCOPE, OBJECT_SCOPE, ONELEVEL_SCOPE	No
attributes	attributes to retrieve	Yes

Delete

Delete Example

```
SELECT x.* FROM (call pm1.native('delete;uid=doe,ou=people,o=teiid.org')) w,
 ARRAYTABLE(w.tuple COLUMNS "updatecount" integer) AS x
```

form the above code, the "**delete**" keyword followed the "DN" string. All the string contents after the "delete;" used as DN.

Create or Update

Create Example

```
SELECT x.* FROM
 (call pm1.native('create;uid=doe,ou=people,o=teiid.org;attributes=one,two,three', 'one', 2, 3.0)) w,
 ARRAYTABLE(w.tuple COLUMNS "update_count" integer) AS x
```

form the above code, the "**create**" keyword followed the "DN" string. All the string contents after the "create;" is used as DN. It also takes one property called "attributes" which is comma separated list of attributes. The values for each attribute is specified as separate argument to the "native" procedure.

Update is similar to "create".

Update Example

```
SELECT x.* FROM
 (call pm1.native('update;uid=doe,ou=people,o=teiid.org;attributes=one,two,three', 'one', 2, 3.0)) w,
 ARRAYTABLE(w.tuple COLUMNS "update_count" integer) AS x
```

LDAP Connector Capabilities Support

LDAP does not provide the same set of functionality as a relational database. The LDAP Connector supports many standard SQL constructs, and performs the job of translating those constructs into an equivalent LDAP search statement. For example, the SQL statement:

```
SELECT firstname, lastname, guid
FROM public_views.people
WHERE
(lastname='Jones' and firstname IN ('Michael', 'John'))
OR
guid > 600000
```

uses a number of SQL constructs, including:

- SELECT clause support
- select individual element support (firstname, lastname, guid)
- FROM support
- WHERE clause criteria support
- nested criteria support
- AND, OR support
- Compare criteria (Greater-than) support
- IN support

The LDAP Connector executes LDAP searches by pushing down the equivalent LDAP search filter whenever possible, based on the supported capabilities. Teiid automatically provides additional database functionality when the LDAP Connector does not explicitly provide support for a given SQL construct. In these cases, the SQL construct cannot be pushed down to the data source, so it will be evaluated in Teiid, in order to ensure that the operation is performed. In cases where certain SQL capabilities cannot be pushed down to LDAP, Teiid pushes down the capabilities that are supported, and fetches a set of data from LDAP. Teiid then evaluates the additional capabilities, creating a subset of the original data set. Finally, Teiid will pass the result to the client. It is useful to be aware of unsupported capabilities, in order to avoid fetching large data sets from LDAP when possible.

LDAP Connector Capabilities Support List

The following capabilities are supported in the LDAP Connector, and will be evaluated by LDAP:

- SELECT queries
- SELECT element pushdown (for example, individual attribute selection)
- AND criteria
- Compare criteria (e.g. <, ⇐, >, >=, =, !=)
- IN criteria
- LIKE criteria.
- OR criteria
- INSERT, UPDATE, DELETE statements (must meet Modeling requirements)

Due to the nature of the LDAP source, the following capability is not supported:

- SELECT queries

The following capabilities are not supported in the LDAP Connector, and will be evaluated by Teiid after data is fetched by the connector:

- Functions
- Aggregates
- BETWEEN Criteria
- Case Expressions
- Aliased Groups
- Correlated Subqueries
- EXISTS Criteria
- Joins
- Inline views
- IS NULL criteria
- NOT criteria
- ORDER BY
- Quantified compare criteria
- Row Offset
- Searched Case Expressions

- Select Distinct
- Select Literals
- UNION
- XA Transactions

Usage

ldap-as-a-datasource quickstart demonstrates using the ldap Translator to access data in OpenLDAP Server. The name of the translator to use in vdb.xml is "translator-ldap", for example:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<vdb name="ldapVDB" version="1">
<model name="HRModel">
<source name="local" translator-name="translator-ldap"
connection-jndi-name="java:/ldapDS"/>
</model>
</vdb>
```

The translator does not provide a connection to the OpenLDAP. For that purpose, Teiid has a JCA adapter that provides a connection to OpenLDAP using the Java Naming API. To define such connector, use the following XML fragment in standalone-teiid.xml. See a example in "<jboss-as>/docs/teiid/datasources/ldap"

```
<resource-adapter id="ldapQS">
<module slot="main" id="org.jboss.teiid.resource-adapter.ldap"/>
<connection-definitions>
<connection-definition
class-name="org.teiid.resource.adapter.ldap.LDAPManagedConnectionFactory"
jndi-name="java:/ldapDS" enabled="true" use-java-context="true"
pool-name="ldapDS">
<config-property name="LdapAdminUserPassword">
redhat
</config-property>
<config-property name="LdapAdminUserDN">
cn=Manager,dc=example,dc=com
</config-property>
<config-property name="LdapUrl">
ldap://localhost:389
</config-property>
</connection-definition>
</connection-definitions>
</resource-adapter>
```

The above defines the translator and connector. For more ways to create the connector see LDAP Data Sources, LDAP translator can derive the metadata based on existing Users/Groups in LDAP Server, user need to define the metadata. For example, you can define a schema using DDL:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<vdb name="ldapVDB" version="1">
<model name="HRModel">
<metadata type="DDL"><![CDATA[
CREATE FOREIGN TABLE HR_Group (
DN string options (nameinsource `dn'),
SN string options (nameinsource `sn'),
UID string options (nameinsource `uid'),
MAIL string options (nameinsource `mail'),
NAME string options (nameinsource `cn')
) OPTIONS(nameinsource `ou=HR,dc=example,dc=com', updatable true);
]]>
```

```
</metadata>
</model>
</vdb>
```

when SELECT operation below executed against table using Teiid will retrieve Users/Groups in LDAP Server:

```
SELECT * FROM HR_Group
```

LDAP Attribute Datatype Support

LDAP providers currently return attribute value types of java.lang.String and byte[], and do not support the ability to return any other attribute value type. The LDAP Connector currently supports attribute value types of java.lang.String only. Therefore, all attributes are modeled using the String datatype in Teiid Designer. 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, since the native type is String.

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. As you deploy the connector in Console, improper configuration can lead to problems when you attempt to start your connector. You can test your LDAP Connector in Teiid Designer prior to Console deployment by submitting queries at modeling time for verification.

LDAP: Console Deployment Issues

The Console shows an Exception That Says Error Synchronizing the Server, If you receive an exception when you synchronize the server and your LDAP Connector is the only service that does not start, it means that there was a problem starting the connector. Verify whether you have correctly typed in your connector properties to resolve this issue.

JCA Resource Adapter

The resource adapter for this translator provided through "LDAP Data Source", Refer to Admin Guide for configuration.

Loopback Translator

The Loopback translator, known by the type name *loopback*, provides a quick testing solution. It supports all SQL constructs and returns default results, with some configurable behavior.

Execution Properties

Name	Description	Default
ThrowError	true to always throw an error	false
RowCount	Rows returned for non-update queries.	1
WaitTime	Wait randomly up to this number of milliseconds with each source query.	0
PollIntervalInMilli	if positive results will be `asynchronously` returned - that is a DataNotAvailableException will be thrown initially and the engine will wait the poll interval before polling for the results.	-1
DelegateName	set to the name of the translator to mimic the capabilities of	

You can also use the Loopback translator to mimic how a real source query would be formed for a given translator (although loopback will still return dummy data that may not be useful for your situation). To enable this behavior, set the DelegateName property to the name of the translator you wish to mimic. For example to disable all capabilities, set the DelegateName property to "jdbc-simple".

JCA Resource Adapter

A source connection is not required for this translator.

Microsoft Excel Translator

The Microsoft Excel Translator, known by the type name *excel*, exposes querying functionality to Excel documents using File Data Sources. Microsoft Excel is a popular spreadsheet software that is used by all the organizations across the globe for simple reporting purposes. This translator provides an easy way read a Excel spreadsheet and provide contents of the spreadsheet in the tabular form that can be integrated with other sources in Teiid.

Note	**"Does it only work on Windows?"** - No, it works on all platforms, including Windows and Linux. This translator uses Apache POI libraries to access the Excel documents which are platform independent.

Usage

The below table describes how Excel translator interprets the data in Excel document into relational terms.

Excel Term	Relational Term
Workbook	schema
Sheet	Table
Row	Row of data
Cell	Column Definition or Data of a column

Excel translator supports "source metadata" feature, where given Excel workbook, it can introspect and build the schema based on the Sheets defined inside it. There are options available for you guide, to be able to detect header columns and data columns in a work sheet to define the correct metadata of a table.

VDB Example

The below shows an example of a VDB, that shows a exposing a Excel Document.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<vdb name="excelvdb" version="1">
    <model name="excel">
        <property name="importer.headerRowNumber" value="1"/>
        <property name="importer.ExcelFileName" value="names.xls"/>
        <source name="connector" translator-name="excel"  connection-jndi-name="java:/fileDS"/>
    </model>
</vdb>
```

"connection-jndi-name" in above represents connection to Excel document. The Excel translator does NOT provide a connection to the Excel Document. For that purpose, Teiid uses File JCA adapter that provides a connection to Excel. To define such connector, see File Data Sources or see an example in "<jboss-as>/docs/teiid/datasources/file". Once you configure both of the above, you can deploy them to Teiid Server and access the Excel Document using JDBC/ODBC/OData protocol.

Designer VDB

If you are using Designer Tooling, to create Excel based VDB

- Create/use a Teiid Designer Model project
- Use "Teiid Connection >> Source Model" importer, create File Data Source using data source creation wizard and use *excel* as translator in the importer. Based on the Excel document relevant relational tables will be created.
- Create a VDB and deploy into Teiid Server and and access the Excel Document using JDBC/ODBC/OData protocol.

Note	**"Headers in Document?"** - If you have headers in the Excel document, you can guide the import process to select the cell headers as the column names in the table creation process. See "Import Properties" section below on defining the "import" properties.

Import Properties

Import properties guide the schema generation part during the deployment of the VDB. This can be used in a native import or while using "Teiid Connection >> Source Model" in Teiid Designer.

Property Name	Description	Default
importer.excelFileName	Defines the name of the Excel Document to import metadata. This can be defined as a file pattern (*.xls), however when defined as pattern all files must be of same format and the translator will choose an arbitrary file to import metadata from. Use file pattern to read data from multiple Excel documents in the same directory, in single file case choose the absolute name.	required
importer.headerRowNumber	Defines the cell header information to be used as column names	optional, default is first data row of sheet
importer.dataRowNumber	Defines the row number where the data rows start	optional, default is first data row of sheet

It is highly recommended that you define all the above importer properties, such that information inside the Excel Document is correctly interpreted.

Note	Purely numerical cells in a column contain containing mixed types will have a string form matching their decimal representation, thus integral values will have .0 appended. If you need the exact text representation, then cell must be a string value which can be forced by putting a single quote ' in front of the numeric text of the cell, or by putting a single space in front of the numeric text.

Translator Extension Properties

Currently there are no Translator Extension properties defined for this translator.

Metadata Extension Properties

Metadata Extension Properties are the properties that are defined on the schema artifacts like Table, Column, Procedure etc, to describe how the translator needs to interact or interpret with source systems. All the properties are defined with namespace 'http://www.teiid.org/translator/excel/2014[http://www.teiid.org/translator/excel/2014\]', which also has a recognized alias 'teiid_excel'.

Property Name	Schema Item Property Belongs To	Description	Mandatory

FILE	Table	Defines Excel Document name or name pattern (*.xls). File pattern can be used to read data from multiple files.	Yes
FIRST_DATA_ROW_NUMBER	Table	Defines the row number where records start in the sheet (applies to every sheet)	optional
CELL_NUMBER	Column of Table	Defines cell number to use for reading data of particular column	Yes

The below shows an example table that is defined using the Extension Metadata Properties.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<vdb name="excelvdb" version="1">
    <model name="excel">
        <source name="connector" translator-name="excel"  connection-jndi-name="java:/fileDS"/>
         <metadata type="DDL"><![CDATA[
             CREATE FOREIGN TABLE Person (
                ROW_ID integer OPTIONS (SEARCHABLE 'All_Except_Like', "teiid_excel:CELL_NUMBER" 'ROW_ID'),
                FirstName string OPTIONS (SEARCHABLE 'Unsearchable', "teiid_excel:CELL_NUMBER" '1'),
                LastName string OPTIONS (SEARCHABLE 'Unsearchable', "teiid_excel:CELL_NUMBER" '2'),
                Age integer OPTIONS (SEARCHABLE 'Unsearchable', "teiid_excel:CELL_NUMBER" '3'),
                CONSTRAINT PK0 PRIMARY KEY(ROW_ID)
             ) OPTIONS ("NAMEINSOURCE" 'Sheet1',"teiid_excel:FILE" 'names.xlsx', "teiid_excel:FIRST_DATA_ROW_NU
MBER" '2')
         ]]> </metadata>
    </model>
</vdb>
```

Note	**"Extended capabilities using ROW_ID column"** - If you define column, that has extension metadata property "CELL_NUMBER" with value "ROW_ID", then that column value contains the row information from Excel document. You can mark this column as Primary Key. You can use this column in SELECT statements with a restrictive set of capabilities including: comparison predicates, IN predicates and LIMIT. All other columns can **not** be used as predicates in a query.

Tip	User does not have to depend upon "source metadata" import, or Designer tool import to create the schema represented by Excel document, they can manually create a source table and add the appropriate extension properties to make a fully functional model. If you introspect the schema model created by the import, it would look like above.

TIP:Currently Excel translator does not support updates.

JCA Resource Adapter

The Teiid specific Excel Resource Adapter does not exist, user should use File JCA adapter with this translator. See File Data Sources for opening a File based connection.

Native Queries

Note	This feature is not applicable for Excel translator.

Direct Query Procedure

Note	This feature is not applicable for Excel translator.

MongoDB Translator

The MongoDB translator, known by the type name *mongodb*, provides a relational view of data that resides in a MongoDB database. This translator is capable of converting Teiid SQL queries into MongoDB based queries. It supports a full range of SELECT, INSERT, UPDATE and DELETE calls.

MongoDB is a document based "schema-less" database with it own query language - it does not map perfectly with relational concepts or the SQL query language. More and more systems are using a MongoDB kind of NOSQL store for scalability and performance. For example, applications like storing audit logs or managing web site data fits well with MongoDB, and does not require using a structural database like Oracle, Postgres etc. MongoDB uses JSON documents as its primary storage unit, and those documents can have additional embedded documents inside the parent document. By using embedded documents it co-locates the related information to achieve de-normalization that typically requires either duplication of data or joins to achieve querying in a relational database.

To make MongoDB work with Teiid the challenge for the MongoDB translator is "How to design a MongoDB store that can achieve the balance between relational and document based storage?" In our opinion the advantages of "schema-less" design are great at development time, not much at runtime except in few special situations. "Schema-less" can also be a problem with migration of application versions and the ability to query and make use of returned information effectively.

Since it is hard and may be impossible in certain situations to derive a schema based on existing the MongoDB collection(s), Teiid approaches the problem in reverse compared to other translators. When working with MongoDB, Teiid requires the user to define the MongoDB schema upfront using Teiid metadata. Since Teiid only allows relational schema as its metadata, the user needs to define their MongoDB schema in relational terms using tables, procedures, and functions. For the purposes of MongoDB, the Teiid metadata has been extended to support extension properties that can be defined on the table to convert it into a MongoDB based document. These extension properties let users define, how a MongoDB document is structured and stored. Based on the relationships (primary-key, foreign-key) defined on a table, and the cardinality (ONE-to-ONE, ONE-to-MANY, MANY-to-ONE) relations between tables are mapped such that related information can be embedded along with the parent document for co-location (see the de-normalization comment above). Thus a relational schema based design, but document based storage in MongoDB. Currently direct mapping of MANY-to-MANY is not supported.

Who is the primary audience for this translator?

The above may not satisfy every user’s needs. The document structure in MongoDB can be more complex than what Teiid can currently define. We hope this will eventually catch up in future versions of Teiid. This is currently designed for:

1. Users that are using relational databases and would like to move/migrate their data to MongoDB to take advantages scaling and performance with out modifying the end user applications currently running.
2. Users that are starting out with MongoDB and do not have experience with MongoDB, but are seasoned SQL developers. This provides a low barrier of entry compared to using MongoDB directly as an application developer.
3. Integrate other enterprise data sources with MongoDB based data.

Usage

The name of the translator to use in vdb.xml is "*mongodb*". For example:

```
<vdb name="nothwind" version="1">
    <model name="northwind">
        <source name="local" translator-name="mongodb" connection-jndi-name="java:/mongoDS"/>
    </model>
```

```
<vdb>
```

The translator does not provide a connection to the MongoDB. For that purpose, Teiid has a JCA adapter that provides a connection to MongoDB using the MongoDB Java Driver. To define such connector, use the following XML fragment in standalone-teiid.xml. See a example in "<jboss-as>/docs/teiid/datasources/mongodb"

```
    <resource-adapters>
        <resource-adapter id="mongodb">
            <module slot="main" id="org.jboss.teiid.resource-adapter.mongodb"/>
            <transaction-support>NoTransaction</transaction-support>
            <connection-definitions>
                <connection-definition class-name="org.teiid.resource.adapter.mongodb.MongoDBManagedConnectionF
actory"
                        jndi-name="java:/mongoDS"
                        enabled="true"
                        use-java-context="true"
                        pool-name="teiid-mongodb-ds">

                      <!-- MongoDB server list (host:port[;host:port...]) -->
                      <config-property name="RemoteServerList">localhost:27017</config-property>
                      <!-- Database Name in the MongoDB -->
                      <config-property name="Database">test</config-property>
                        <!--
                            Uncomment these properties to supply user name and password
                        <config-property name="Username">user</config-property>
                        <config-property name="Password">user</config-property>
                        -->
                </connection-definition>
            </connection-definitions>
        </resource-adapter>
    </resource-adapters>
```

The above defines the translator and connector. For more ways to create the connector see MongoDB Data Sources. MongoDB translator can derive the metadata based on existing document collections in some scenarios, however when working with complex documents the interpretation of metadata may be inaccurate, in those situations the user MUST define the metadata. For example, you can define a schema using DDL:

```
<vdb name="nothwind" version="1">
    <model name="northwind">
        <source name="local" translator-name="mongodb" connection-jndi-name="java:/mongoDS"/>
            <metadata type="DDL"><![CDATA[
                CREATE FOREIGN TABLE  Customer (
                    customer_id integer,
                    FirstName varchar(25),
                    LastName varchar(25)
                ) OPTIONS(UPDATABLE 'TRUE');
            ]]> </metadata>
    </model>
<vdb>
```

when INSERT operation below executed against table using Teiid,

```
    INSERT INTO Customer(customer_id, FirstName, LastName) VALUES (1, 'John', 'Doe');
```

MongoDB translator will create a below document in the MongoDB

```
{
  _id: ObjectID("509a8fb2f3f4948bd2f983a0"),
  customer_id: 1,
  FirstName: "John",
  LastName: "Doe"
```

```
}
```

If a PRIMARY KEY is defined on the table as

```
    CREATE FOREIGN TABLE  Customer (
        customer_id integer PRIMARY KEY,
        FirstName varchar(25),
        LastName varchar(25)
    ) OPTIONS(UPDATABLE 'TRUE');
```

then that column name is automatically used as "_id" field in the MongoDB collection, then document structure is stored in the MongoDB as

```
{
  _id: 1,
  FirstName: "John",
  LastName: "Doe"
}
```

If you defined the composite PRIMARY KEY on Customer table as

```
    CREATE FOREIGN TABLE  Customer (
        customer_id integer,
        FirstName varchar(25),
        LastName varchar(25),
        PRIMARY KEY (FirstName, LastName)
    ) OPTIONS(UPDATABLE 'TRUE');
```

the document structure will be

```
{
  _id: {
         FirstName: "John",
         LastName:  "Doe"
       },
  customer_id: 1,
}
```

Data Types

MongoDB translator supports automatic mapping of Teiid data types into MongoDB data types, including the support for Blobs, Clobs and XML. The LOB support is based on GridFS in MongoDB. Arrays are in the form of

```
{
  _id: 1,
  FirstName: "John",
  LastName: "Doe"
  Score: [89, "ninety", 91.0]
}
```

are supported. User can get individual items in the array using function array_get, or can transform the array into tabular structure using ARRATTABLE.

Note	Note that even though embedded documents can also be in arrays, the handling of embedded documents is different from array with scalar values.

Regular Expressions, MongoDB::Code, MongoDB::MinKey, MongoDB::MaxKey, MongoDB::OID is not currently supported.

Extension Metadata Properties To Build Complex Documents

Using the above DDL or any other metadata facility, a user can map a table in a relational store into a document in MongoDB, however to make effective use of MongoDB, you need to be able to build complex documents, that can co-locate related information, so that data can queried in a single MongoDB query. Otherwise, since MongoDB does not support join relationships like relational database, you need to issue multiple queries to retrieve and join data manually. The power of MongoDB comes from its "embedded" documents and its support of complex data types like arrays and use of the aggregation framework to be able to query them. This translator provides way to achieve that goals.

When you do not define the complex embedded documents in MongoDB, Teiid can step in for join processing and provide that functionality, however if you want to make use of the power of MongoDB itself in querying the data and avoid bringing the unnecessary data and improve performance, you need to look into building these complex documents.

MongoDB translator defines two additional metadata properties along with other Teiid metadata properties to aid in building the complex "embedded" documents. You can use the following metadata properties in your DDL.

- **teiid_mongo:EMBEDDABLE** - Means that data defined in this table is allowed to be included as an "embeddable" document in **any** parent document. The parent document is referenced by the foreign key relationships. In this scenario, Teiid maintains more than one copy of the data in MongoDB store, one in its own collection and also a copy in each of the parent tables that have relationship to this table. You can even nest embeddable table inside another embeddable table with some limitations. Use this property on table, where table can exist, encompass all its relations on its own. For example, a "Category" table that defines a "Product"'s category is independent of Product, which can be embeddable in "Products" table.
- **teiid_mongo:MERGE** - Means that data of this table is merged with the defined parent table. There is only a single copy of the data that is embedded in the parent document. Parent document is defined using the foreign key relationships.

Using the above properties and FOREIGN KEY relationships, we will illustrate how to build complex documents in MongoDB.

Note	**Usage** - Please note a given table can contain either the "teiid_mongo:EMBEDDABLE" property or the "teiid_mongo:MERGE" property defining the type of nesting in MongoDB. A table is not allowed to have both properties.

2-ONE Mapping

If your current DDL structure representing ONE-2-ONE relationship is like

```
    CREATE FOREIGN TABLE  Customer (
        CustomerId integer PRIMARY KEY,
        FirstName varchar(25),
        LastName varchar(25)
    ) OPTIONS(UPDATABLE 'TRUE');

    CREATE FOREIGN TABLE Address (
        CustomerId integer,
        Street varchar(50),
        City varchar(25),
        State varchar(25),
        Zipcode varchar(6),
        FOREIGN KEY (CustomerId) REFERENCES Customer (CustomerId)
     ) OPTIONS(UPDATABLE 'TRUE');
```

by default, this will produce two different collections in MongoDB, like with sample data it will look like

```
Customer
{
```

```
  _id: 1,
  FirstName: "John",
  LastName: "Doe"
}

Address
{
  _id: ObjectID("..."),
   CustomerId: 1,
   Street: "123 Lane"
   City: "New York",
   State: "NY"
   Zipcode: "12345"
}
```

You can enhance the storage in MongoDB to a single collection by using "teiid_mongo:MERGE' extension property on the table's OPTIONS clause

```
    CREATE FOREIGN TABLE  Customer (
        CustomerId integer PRIMARY KEY,
        FirstName varchar(25),
        LastName varchar(25)
    ) OPTIONS(UPDATABLE 'TRUE');

    CREATE FOREIGN TABLE Address (
        CustomerId integer PRIMARY KEY,
        Street varchar(50),
        City varchar(25),
        State varchar(25),
        Zipcode varchar(6),
        FOREIGN KEY (CustomerId) REFERENCES Customer (CustomerId)
     ) OPTIONS(UPDATABLE 'TRUE', "teiid_mongo:MERGE" 'Customer');
```

this will produce single collection in MongoDB, like

```
Customer
{
  _id: 1,
  FirstName: "John",
  LastName: "Doe",
  Address:
     {
        Street: "123 Lane",
        City: "New York",
        State: "NY",
        Zipcode: "12345"
     }
}
```

With the above both tables are merged into a single collection that can be queried together using the JOIN clause in the SQL command. Since the existence of child/additional record has no meaning with out parent table using the "*teiid_mongo:MERGE*" extension property is right choice in this situation.

Note	Note that the Foreign Key defined on child table, must refer to Primary Keys on both parent and child tables to form a One-2-One relationship.

2-MANY Mapping.

Typically there can be more than two (2) tables involved in this relationship. If MANY side is only associated **single** table, then use "teiid_mongo:MERGE" property on MANY side of table and define ONE as the parent. If associated with more than single table then use "teiid_mongo:EMBEDDABLE".

For example if you have DDL like

```
    CREATE FOREIGN TABLE  Customer (
        CustomerId integer PRIMARY KEY,
        FirstName varchar(25),
        LastName varchar(25)
    ) OPTIONS(UPDATABLE 'TRUE');

    CREATE FOREIGN TABLE  Order (
        OrderID integer PRIMARY KEY,
        CustomerId integer,
        OrderDate date,
        Status integer,
        FOREIGN KEY (CustomerId) REFERENCES Customer (CustomerId)
    ) OPTIONS(UPDATABLE 'TRUE');
```

in the above a Single Customer can have MANY Orders. There are two options to define the how we store the MongoDB document. If in your schema, the Customer table's CustomerId is **only** referenced in Order table (i.e. Customer information used for only Order purposes), you can use

```
    CREATE FOREIGN TABLE  Customer (
        CustomerId integer PRIMARY KEY,
        FirstName varchar(25),
        LastName varchar(25)
    ) OPTIONS(UPDATABLE 'TRUE');

    CREATE FOREIGN TABLE  Order (
        OrderID integer PRIMARY KEY,
        CustomerId integer,
        OrderDate date,
        Status integer,
        FOREIGN KEY (CustomerId) REFERENCES Customer (CustomerId)
    ) OPTIONS(UPDATABLE 'TRUE', "teiid_mongo:MERGE" 'Customer');
```

that will produce a single document for Customer table like

```
{
  _id: 1,
  FirstName: "John",
  LastName: "Doe",
  Order:
  [
     {
       _id: 100,
       OrderDate: ISODate("2000-01-01T06:00:00Z")
       Status: 2
     },
     {
       _id: 101,
       OrderDate: ISODate("2001-03-06T06:00:00Z")
       Status: 5
     }
     ...
   ]
}
```

If Customer table is referenced in more tables other than Order table, then use "teiid_mongo:EMBEDDABLE" property

```
    CREATE FOREIGN TABLE Customer (
        CustomerId integer PRIMARY KEY,
        FirstName varchar(25),
        LastName varchar(25)
    ) OPTIONS(UPDATABLE 'TRUE', "teiid_mongo:EMBEDDABLE" 'TRUE');
```

```
    CREATE FOREIGN TABLE Order (
        OrderID integer PRIMARY KEY,
        CustomerId integer,
        OrderDate date,
        Status integer,
        FOREIGN KEY (CustomerId) REFERENCES Customer (CustomerId)
    ) OPTIONS(UPDATABLE 'TRUE');

    CREATE FOREIGN TABLE Comments (
        CommentID integer PRIMARY KEY,
        CustomerId integer,
        Comment varchar(140),
        FOREIGN KEY (CustomerId) REFERENCES Customer (CustomerId)
    ) OPTIONS(UPDATABLE 'TRUE');
```

This creates three different collections in MongoDB.

```
Customer
{
  _id: 1,
  FirstName: "John",
  LastName: "Doe"
}

Order
{
  _id: 100,
  CustomerId: 1,
  OrderDate: ISODate("2000-01-01T06:00:00Z")
  Status: 2
  Customer:
   {
     FirstName: "John",
     LastName: "Doe"
   }
}

Comment
{
  _id: 12,
  CustomerId: 1,
  Comment: "This works!!!"
  Customer:
   {
     FirstName: "John",
     LastName: "Doe"
   }
}
```

Here as you can see the Customer table contents are embedded along with other table's data where they were referenced. This creates duplicated data where multiple of these embedded documents are managed automatically in the MongoDB translator.

Note	All the SELECT, INSERT, DELETE operations that are generated against the tables with "teiid_mongo:EMBEDDABLE" property are atomic, except for UPDATES, as there can be multiple operations involved to update all the copies. Since there are no transactions in MongoDB, Teiid plans to provide automatic compensating transaction framework around this in future releases TEIID-2957.

MANY-2-ONE Mapping.

This is same as ONE-2-MANY, see above to define relationships.

Note	A parent table can have multiple "embedded" and as well as "merge" documents inside it, it not limited so either one or other. However, please note that MongoDB imposes document size is limited can not exceed 16MB.

MANY-2-MANY Mapping.

This can also mapped with combination of "teiid_mongo:MERGE" and "teiid_mongo:EMBEDDABLE" properties (partially). For example if DDL looks like

```
CREATE FOREIGN TABLE Order (
    OrderID integer PRIMARY KEY,
    OrderDate date,
    Status integer
) OPTIONS(UPDATABLE 'TRUE');

CREATE FOREIGN TABLE OrderDetail (
    OrderID integer,
    ProductID integer,
    PRIMARY KEY (OrderID,ProductID),
    FOREIGN KEY (OrderID) REFERENCES Order (OrderID),
    FOREIGN KEY (ProductID) REFERENCES Product (ProductID)
) OPTIONS(UPDATABLE 'TRUE');

CREATE FOREIGN TABLE Products (
   ProductID integer PRIMARY KEY,
   ProductName varchar(40)
) OPTIONS(UPDATABLE 'TRUE');
```

you modify the DDL like below, to have

```
CREATE FOREIGN TABLE Order (
    OrderID integer PRIMARY KEY,
    OrderDate date,
    Status integer
) OPTIONS(UPDATABLE 'TRUE');

CREATE FOREIGN TABLE OrderDetail (
    OrderID integer,
    ProductID integer,
    PRIMARY KEY (OrderID,ProductID),
    FOREIGN KEY (OrderID) REFERENCES Order (OrderID),
    FOREIGN KEY (ProductID) REFERENCES Product (ProductID)
) OPTIONS(UPDATABLE 'TRUE', "teiid_mongo:MERGE" 'Order');

CREATE FOREIGN TABLE Products (
   ProductID integer PRIMARY KEY,
   ProductName varchar(40)
) OPTIONS(UPDATABLE 'TRUE',  "teiid_mongo:EMBEDDABLE" 'TRUE');
```

That will produce a document like

```
{
   _id : 10248,
   OrderDate : ISODate("1996-07-04T05:00:00Z"),
   Status : 5
   OrderDetails : [
     {
       _id : {
               OrderID : 10248,
               ProductID : 11
               Products : {
                  ProductID: 11
                  ProductName: "Hammer"
               }
       }
     },
     {
```

```
        _id : {
          OrderID : 10248,
          ProductID : 14
          Products : {
              ProductID: 14
              ProductName: "Screw Driver"
          }
        }
      }
    ]
}

Products
{
    {
      ProductID: 11
      ProductName: "Hammer"
    }
    {
      ProductID: 14
      ProductName: "Screw Driver"
    }
}
```

Limitations

- Currently nested embedding of documents has limited support due to capabilities of handling nested arrays is limited in the MongoDB. Nesting of "EMBEDDALBLE" property with multiple levels is OK, however more than two levels with MERGE is not recommended. Also, you need to be caution about not exceeding the document size of 16 MB for single row, so deep nesting is not recommended.
- JOINS between related tables, MUST have used either of "EMBEDDABLE" or "MERGE" property, otherwise the query will result in error. In order for Teiid to correctly plan and support the JOINS, in the case that any two tables are **NOT** embedded in each other, use *allow-joins=false* property on the Foreign Key that represents the relation. For example:

```
CREATE FOREIGN TABLE  Customer (
    CustomerId integer PRIMARY KEY,
    FirstName varchar(25),
    LastName varchar(25)
) OPTIONS(UPDATABLE 'TRUE');

CREATE FOREIGN TABLE  Order (
    OrderID integer PRIMARY KEY,
    CustomerId integer,
    OrderDate date,
    Status integer,
    FOREIGN KEY (CustomerId) REFERENCES Customer (CustomerId) OPTIONS (allow-join 'FALSE')
) OPTIONS(UPDATABLE 'TRUE');
```

with the example above, Teiid will create two collections, however when user issues query such as

```
SELECT OrderID, LastName FROM Order JOIN Customer ON Order.CustomerId = Customer.CustomerId;
```

instead of resulting in error, the JOIN processing will happen in the Teiid engine, without the above property it will result in an error.

When you use above properties and carefully design the MongoDB document structure, Teiid translator can intelligently collate data based on their co-location and take advantage of it while querying.

Geo Spatial function support

MongoDB translator supports geo spatial query operators in the "WHERE" clause, when the data is stored in the GeoJSon format in the MongoDB Document. The supported functions are

```
CREATE FOREIGN FUNCTION geoIntersects (columnRef string,  type string, coordinates double[][]) RETURNS boolean;
CREATE FOREIGN FUNCTION geoWithin (ccolumnRef string,  type string, coordinates double[][]) RETURNS boolean;
CREATE FOREIGN FUNCTION near (ccolumnRef string,  coordinates double[], maxdistance integer) RETURNS boolean;
CREATE FOREIGN FUNCTION nearSphere (ccolumnRef string, coordinates double[], maxdistance integer) RETURNS boole
an;
CREATE FOREIGN FUNCTION geoPolygonIntersects (ref string, north double, east double, west double, south double)
RETURNS boolean;
CREATE FOREIGN FUNCTION geoPolygonWithin (ref string, north double, east double, west double, south double) RET
URNS boolean;
```

a sample query looks like

```
SELECT loc FROM maps where mongo.geoWithin(loc, 'LineString', ((cast(1.0 as double), cast(2.0 as double)), (cast
(1.0 as double), cast(2.0 as double))))
```

Same functions using built-in Geometry type (the above functions will be deprecated and removed in future versions)

```
CREATE FOREIGN FUNCTION geoIntersects (columnRef string,  geo geometry) RETURNS
boolean;
CREATE FOREIGN FUNCTION geoWithin (ccolumnRef string,  geo geometry) RETURNS
boolean;
CREATE FOREIGN FUNCTION near (ccolumnRef string, geo geometry, maxdistance integer)
RETURNS boolean;
CREATE FOREIGN FUNCTION nearSphere (ccolumnRef string, geo geometry, maxdistance
integer) RETURNS boolean;
CREATE FOREIGN FUNCTION geoPolygonIntersects (ref string, geo geometry) RETURNS
boolean;
CREATE FOREIGN FUNCTION geoPolygonWithin (ref string, geo geometry) RETURNS
boolean;
```

a sample query looks like

```
SELECT loc FROM maps where mongo.geoWithin(loc, ST_GeomFromGeoJSON('{"coordinates":[[1,2],[3,4]],"type":"Polygo
n"}'))
```

There are various "st_geom.." methods are available in the Geo Spatial function library in Teiid.

Capabilities

MongoDB translator designed on top of the MongoDB aggregation framework, use of MongoDB version that supports this framework is mandatory. Apart from SELECT queries, this translator also supports INSERT, UPDATE and DELETE queries.

This translator supports

- grouping
- matching
- sorting

- filtering
- limit
- support for LOBs using GridFS
- Composite primary and foreign keys.

Note	**example** - For a full example see https://github.com/teiid/teiid/blob/master/connectors/translator-mongodb/src/test/resources/northwind.ddl

Native Queries

MongoDB source procedures may be created using the teiid_rel:native-query extension - see Parameterizable Native Queries. The procedure will invoke the native-query similar to a direct procedure call with the benefits that the query is predetermined and that result column types are known, rather than requiring the use of ARRAYTABLE or similar functionality.

Direct Query Procedure

This feature is turned off by default because of the security risk this exposes to execute any command against the source. To enable this feature, override the execution property called _SupportsDirectQueryProcedure to true.

By default the name of the procedure that executes the queries directly is called **native**. Override the execution property _DirectQueryProcedureName to change it to another name.

The MongoDB translator provides a procedure to execute any ad-hoc aggregate query directly against the source without Teiid parsing or resolving. Since the metadata of this procedure's results are not known to Teiid, they are returned as an object array containing single blob at array location one(1). This blob contains the JSON document. XMLTABLE can be used construct tabular output for consumption by client applications.

Example MongoDB Direct Query

```
     select x.* from TABLE(call native('city;{$match:{"city":"FREEDOM"}}')) t,
          xmltable('/city' PASSING JSONTOXML('city', cast(array_get(t.tuple, 1) as BLOB)) COLUMNS city string,
state string) x
```

In the above example, a collection called "city" is looked up with filter that matches the "city" name with "FREEDOM", using "native" procedure and then using the nested tables feature the output is passed to a XMLTABLE construct, where the output from the procedure is sent to a JSONTOXML function to construct a XML then the results of that are exposed in tabular form.

The direct query MUST be in the format

```
      "collectionName;{$pipeline instr}+"
```

From Teiid 8.10, MongoDB translator also allows to execute Shell type java script commands like remove, drop, createIndex. For this the command needs to be in format

```
      "$ShellCmd;collectionName;operationName;{$instr}+"
```

and example looks like

```
    "$ShellCmd;MyTable;remove;{ qty: { $gt: 20 }}"
```

Object Translator

The Object translator, known by the name of *map-cache*, is a bridge for reading and writing java objects from external sources (i.e., Map cache) and delivering them to the engine for processing. And to assist in providing that bridge, the OBJECTTABLE function is an alternaitive means for transforming complex java objects into rows and columns.

Search Capabilities

Supports a local cache that is of type Map and it uses *key* searching on the map to find objects.

Supported Capabilities

The following are the connector capabilities when Key Searching is used:

- SELECT command
- CompareCriteria - only EQ
- InCriteria
- Insert, Update and Delete

Usage

- Retrieve objects from a cache and transform into rows and columns.
- Perform inserts, updates and deletes to the cache

Properties

Object translator is capable of returning to the client application the java object stored in the cache. To enable the translator to automatically include the object as a column in the metadata, use the following Import Properties.

Import Properties

Property Name	Description	Required	Default
ClassObjectColumn	If true, and when the translator provides the metadata, a column of object data type will be created that represents the stored object in the cache	false	false

Metadata

Requirements for Defining Table for Root Class

Table level requirements

- The table for the root class, must have a primary key defined, which must map to an attribute in the class.
- The table name (or the name in source) must match the name of the Pojo class name. This is how the table is matched to the registered class in the cache.

Note	The primary key data type for the attribute in the class must match the cache key data type.

Column level requirements

- The class object column that represents the cached object should have a name in source of 'this'. All other columns will have their name in source (which defaults to the column name) interpreted as the path to the column value from the primary object.
- All columns that are not the primary key nor covered by a lucene index should be marked as SEARCHABLE 'Unsearchable'.
- Attributes defined as repeatable (i.e., collections, arrays, or map) or a container class, can be supported as 1-to-* relationship, and will have corresponding registered class (if they are to be searched).

Requirements for Defining Child Table in a Relationship

A relationship needs to be defined if the primary class contains a complex object type that you want to query.

Supported relationships

- 1-to-1 - must have getter/setter for a single object
- 1-to-many - must have getter/setter for a Collection, Map or Array type.

Table level requirements

- The table name (or the name in source) must match the name of the Pojo class storing the child information. This is how the table is matched to the registered class(s) in the cache.
- A primary key is required if updates are to be performed
- The child table must have a foreign key that maps to the parent table primary key.
- The name in source for the foreign key is the name of the parent class method to access the child objects.

Example for Defining Metadata

The following is an example of a Person that can have Phones. This demonstrates how to define the primary class and a relationship using DDL.

```
public class Person {

   public String name;
   public int id;
   public String email;

   public List<PhoneNumber> phones;

   public String getName() {
      return name;
   }

   public void setName(String name) {
      this.name = name;
   }
```

```
   public int getId() {
      return id;
   }

   public void setId(int id) {
      this.id = id;
   }

   public String getEmail() {
      return email;
   }

   public void setEmail(String email) {
      this.email = email;
   }

   public List<PhoneNumber> getPhones() {
      return phones;
   }

   public void setPhones(List<PhoneNumber> phones) {
      this.phones = phones;
   }

}

public class PhoneNumber {

   private String number;
   private String type;

   public String getNumber() {
      return number;
   }

   public void setNumber(String number) {
      this.number = number;
   }

   public String getType() {
      return type;
   }

   public void setType(String type) {
      this.type = type;
   }
}
```

Note, this also shows a container class, PhoneNumber, as an example of the foreign key that's defines the relationship.

```
<vdb name="PersonVDB" version="1">
    <model name="PersonModel" visible="false">
        <source name="objsource" translator-name="map-cache" connection-jndi-name="java:cache-jndi"/>
        <metadata type="DDL"><![CDATA[

                CREATE FOREIGN TABLE Person (
                        PersonObject object OPTIONS (NAMEINSOURCE 'this', SELECTABLE FALSE, UPDATABLE FALSE, SE
ARCHABLE 'Unsearchable', NATIVE_TYPE 'org.jboss.as.quickstarts.datagrid.hotrod.query.domain.Person'),
                        id integer NOT NULL OPTIONS (NAMEINSOURCE 'id', SEARCHABLE 'Searchable', NATIVE_TYPE 'i
nt'),
                        name string OPTIONS (NAMEINSOURCE 'name', SEARCHABLE 'Searchable', NATIVE_TYPE 'java.la
ng.String'),
                        email string OPTIONS (NAMEINSOURCE 'email', SEARCHABLE 'Searchable', NATIVE_TYPE 'java.
lang.String'),
                        CONSTRAINT PK_ID PRIMARY KEY(id)
                ) OPTIONS (UPDATABLE TRUE);
```

```
                CREATE FOREIGN TABLE PhoneNumber (
                        id integer NOT NULL OPTIONS (NAMEINSOURCE 'id', SELECTABLE FALSE, UPDATABLE FALSE, SEAR
CHABLE 'Searchable', NATIVE_TYPE 'int'),
                        number string OPTIONS (NAMEINSOURCE 'phone.number', SEARCHABLE 'Searchable', NATIVE_TYP
E 'java.lang.String'),
                        type string OPTIONS (NAMEINSOURCE 'phone.type', SEARCHABLE 'Unsearchable', NATIVE_TYPE
'java.lang.Enum'),
                        CONSTRAINT FK_PERSON FOREIGN KEY(id) REFERENCES Person (id) OPTIONS (NAMEINSOURCE 'phon
es')
                ) OPTIONS (UPDATABLE TRUE);

        ]]> </metadata>
    </model>

</vdb>
```

This metadata could also be defined by using the Teiid Designer.

JCA Resource Adapter

OData Translator

The OData translator, known by the type name *"odata"* exposes the OData V2 and V3 data sources and uses the Teiid WS resource adapter for making web service calls. This translator is extension of *Web Services Translator*.

Note	**What is Odata** - The Open Data Protocol (OData) Web protocol is for querying and updating data that provides a way to unlock your data and free it from silos that exist in applications today. OData does this by applying and building upon Web technologies such as HTTP, Atom Publishing Protocol (AtomPub) and JSON to provide access to information from a variety of applications, services, and stores. OData is being used to expose and access information from a variety of sources including, but not limited to, relational databases, file systems, content management systems and traditional Web sites.

Using this specification from OASIS group, with the help from the OData4J framework, Teiid maps OData entities into relational schema. Teiid supports reading of CSDL (Conceptual Schema Definition Language) from the OData endpoint provided and converts the OData schema into relational schema. The below table shows the mapping selections in OData Translator from CSDL document

OData	Mapped to Relational Entity
EntitySet	Table
FunctionImport	Procedure
AssosiationSet	Foreign Keys on the Table*
ComplexType	ignored**

- A Many to Many association will result in a link table that can not be selected from, but can be used for join purposes.
  - When used in Functions, an implicit table is exposed. When used to define a embedded table, all the columns will be in-lined

All CRUD operations will be appropriately mapped to the resulting entity based on the SQL submitted to the OData translator.

Usage

Usage of a OData source is similar a JDBC translator. The metadata import is supported through the translator, once the metadata is imported from source system and exposed in relational terms, then this source can be queried as if the EntitySets and Function Imports were local to the Teiid system.

Execution Properties

Name	Description	Default
DatabaseTimeZone	The time zone of the database. Used when fetchings date, time, or timestamp values	The system default time zone
SupportsOdataCount	Supports $count	true
SupportsOdataFilter	Supports $filter	true
SupportsOdataOrderBy	Supports $orderby	true

SupportsOdataSkip	Supports $skip	true
SupportsOdataTop	Supports $top	true

Importer Properties

Name	Description	Default
schemaNamespace	Namespace of the schema to import	null
entityContainer	Entity Container Name to import	default container

Example importer settings to only import tables and views from NetflixCatalog.

```
<property name="importer.schemaNamespace" value="System.Data.Objects"/>
<property name="importer.schemaPattern" value="NetflixCatalog"/>
```

Note	**OData Server is not fully compatible** - Sometimes it's possible that the odata server you are querying does not fully implement all OData specification features. If your OData implementation does not support a certain feature, then turn off the corresponding capability using "execution Properties", so that Teiid will not pushdown invalid queries to the translator. For example, to turn off $filter you add following to your vdb.xml

```
    <translator name="odata-override" type="odata">
    <property name="SupportsOdataFilter" value="false"/>
    </translator>
```

then use "odata-override" as the translator name on your source model.

Tip	**Native Queries** - Native or direct query execution is not supported through OData translator. However, user can use Web Services Translator's *invokehttp* method directly to issue a Rest based call and parse results using SQLXML.

Tip	**Want to use as Server?** - Teiid can not only consume OData based data sources, but it can expose any data source as an Odata based webservice. For more information see OData Support.

JCA Resource Adapter

The resource adapter for this translator is a Web Service Data Source.

OData V4 Translator

The OData V4 translator, known by the type name *"odata4"* exposes the OData Version 4 data sources and uses the Teiid WS resource adapter for making web service calls. This translator is extension of *Web Services Translator*.

Note	**What is Odata** - The Open Data Protocol (OData) Web protocol is for querying and updating data that provides a way to unlock your data and free it from silos that exist in applications today. OData does this by applying and building upon Web technologies such as HTTP, Atom Publishing Protocol (AtomPub) and JSON to provide access to information from a variety of applications, services, and stores. OData is being used to expose and access information from a variety of sources including, but not limited to, relational databases, file systems, content management systems and traditional Web sites.

Using this specification from OASIS group, with the help from the Olingo framework, Teiid maps OData V4 CSDL (Conceptual Schema Definition Language) document from the OData endpoint provided and converts the OData metadata into Teiid's relational schema. The below table shows the mapping selections in OData V4 Translator from CSDL document

OData	Mapped to Relational Entity
EntitySet	Table
EntityType	Table see [1]
ComplexType	Table see [2]
FunctionImport	Procedure [3]
ActionImport	Procedure [3]
NavigationProperties	Table [4]

[1] Only if the EntityType is exposed as the EntitySet in the Entity Container [2] Only if the complex type is used as property in the exposed EntitySet. This table will be designed as child table with foreign key [1 to 1] or [1 to many] relationship to the parent [3] If the return type is EntityType or ComplexType, the procedure is designed to return a table [4] Navigation properties are exposed as tables. The table will be created with foreign key relationship to the parent.

All CRUD operations will be appropriately mapped to the resulting entity based on the SQL submitted to the OData translator.

Usage

Usage of a OData source is similar a JDBC translator. The metadata import is supported through the translator, once the metadata is imported from source system and exposed in relational terms, then this source can be queried as if the EntitySets, Function Imports and Action Imports were local to the Teiid system.

The below is sample VDB that can read metadata service from TripPin service on http://odata.org site.

```
<vdb name="trippin" version="1">
    <model name="trippin">
         <source name="odata4" translator-name="odata4" connection-jndi-name="java:/tripDS"/>
    </model>
</vdb>
```

The required resource-adapter configuration will look like

```
<resource-adapter id="trippin">
    <module slot="main" id="org.jboss.teiid.resource-adapter.webservice"/>
    <transaction-support>NoTransaction</transaction-support>
    <connection-definitions>
        <connection-definition class-name="org.teiid.resource.adapter.ws.WSManagedConnectionFactory" jndi-name=
"java:/tripDS" enabled="true" use-java-context="true" pool-name="teiid-trip-ds">
            <config-property name="EndPoint">
                http://services.odata.org/V4/(S(va3tkzikqbtgu1ist44bbft5))/TripPinServiceRW
            </config-property>
        </connection-definition>
    </connection-definitions>
</resource-adapter>
```

Once you configure above resource-adapter and deploy the VDB successfully, then you can connect to the VDB deployed using Teiid JDBC driver and issue SQL statements like

```
SELECT * FROM trippin.People;
SELECT * FROM trippin.People WHERE UserName = 'russelwhyte';
SELECT * FROM trippin.People p INNER JOIN trippin.People_Friends pf ON p.UserName = pf.People_UserName; (note t
hat People_UserName is implicitly added by Teiid metadata)
EXEC GetNearestAirport(lat, lon) ;
```

Configuration of Translator

Execution Properties

Execution properties extend/limit the functionality of the translator based on the physical source capabilities. Sometimes default properties need to adjusted for proper execution of the translator.

Execution Properties

Name	Description	Default
SupportsOdataCount	Supports $count	true
SupportsOdataFilter	Supports $filter	true
SupportsOdataOrderBy	Supports $orderby	true
SupportsOdataSkip	Supports $skip	true
SupportsOdataTop	Supports $top	true
SupportsUpdates	Supports INSERT/UPDATE/DELETE	true

Sometimes it’s possible that the odata server you are querying does not fully implement all OData specification features. If your OData implementation does not support a certain feature, then turn off the corresponding capability using "execution Properties", so that Teiid will not pushdown invalid queries to the translator. For example, to turn off $filter you add following to your vdb.xml

```
<translator name="odata-override" type="odata">
     <property name="SupportsOdataFilter" value="false"/>
</translator>
```

then use "odata-override" as the translator name on your source model.

Importer Properties

Importer properties define the behavior of the translator during the metadata import from the physical source.

Importer Properties

Name	Description	Default
schemaNamespace	Namespace of the schema to import	null

Example importer settings to only import tables and views from Trippin service exposed on odata.org

```
<property name="importer.schemaNamespace" value="Microsoft.OData.SampleService.Models.TripPin"/>
```

You can leave this property undefined, as if it does not find one configured the translator will select the default name of the EntityContainer.

JCA Resource Adapter

The resource adapter for this translator is a Web Service Data Source.

Note	**Native Queries** - Native or direct query execution is not supported through OData translator. However, user can use Web Services Translator's *invokehttp* method directly to issue a Rest based call and parse results using SQLXML.

Note	**Want to use as OData Server?** - Teiid can not only consume OData based data sources, but it can expose any data source as an OData based webservice. For more information see OData Support.

Swagger Translator

The Swagger translator, known by the type name *"swagger"* exposes the Swagger data sources in realational concepts and uses the Teiid WS resource adapter for making web service calls.

Note	**What is Swagger** - http://swagger.io/ [OpenAPI Specification (Swagger)] Swagger is a simple yet powerful representation of your RESTful API. With the largest ecosystem of API tooling on the planet, thousands of developers are supporting Swagger in almost every modern programming language and deployment environment. With a Swagger-enabled API, you get interactive documentation, client SDK generation and discoverability.

Starting January 1st 2016 the Swagger Specification has been donated to the Open API Initiative (OAI) and has been renamed to the OpenAPI Specification.

Usage

Usage of a Swagger source is similar any other translator in Teiid. The metadata import is supported through the translator, the metadata is imported from source system's swagger.json file and then API from this file is exposed as stored procedures in Teiid, then source system can be queried by executing these stored procedures in Teiid system.

The below is sample VDB that can read metadata from Petstore reference service on http://petstore.swagger.io/ site.

```
<vdb name="petstore" version="1">
    <model visible="true" name="m">
        <source name="s" translator-name="swagger" connection-jndi-name="java:/swagger"/>
    </model>
</vdb>
```

The required resource-adapter configuration will look like

```
<resource-adapter id="swagger">
    <module slot="main" id="org.jboss.teiid.resource-adapter.webservice"/>
    <transaction-support>NoTransaction</transaction-support>
    <connection-definitions>
        <connection-definition class-name="org.teiid.resource.adapter.ws.WSManagedConnectionFactory" jndi-name=
"java:/swagger" enabled="true" use-java-context="true" pool-name="teiid-swagger-ds">
            <config-property name="EndPoint">
                http://petstore.swagger.io/v2
            </config-property>
        </connection-definition>
    </connection-definitions>
</resource-adapter>
```

Once you configure above resource-adapter and deploy the VDB successfully, then you can connect to the VDB deployed using Teiid JDBC driver and issue SQL statements like

```
EXEC findPetsByStatus(('sold',))
EXEC getPetById(1461159803)
EXEC deletePet('', 1461159803)
```

Configuration of Translator

Execution Properties

Execution properties extend/limit the functionality of the translator based on the physical source capabilities. Sometimes default properties need to adjusted for proper execution of the translator.

Execution Properties

none

Importer Properties

Importer properties define the behavior of the translator during the metadata import from the physical source.

Importer Properties

Name	Description	Default
useDefaultHost	Use default host specified in the Swagger file; Defaults to true, when false uses the endpoint in the resource-adapter	true
preferredScheme	Preferred Scheme to use when Swagger file supports multiple invocation schemes like http, https	null
preferredProduces	Preferred Accept MIME type header, this should be one of the Swagger 'produces' types;	application/json
preferredConsumes	Preferred Content-Type MIME type header, this should be one of the Swagger 'consumer' types;	application/json

Example importer settings to avoid calling host defined on the swagger.json file

```
<property name="importer.useDefaultHost" value="false"/>
```

JCA Resource Adapter

The resource adapter for this translator is a Web Service Data Source.

Note	**Native Queries** - Native or direct query execution is not supported through Swagger translator. However, user can use Web Services Translator's *invokehttp* method directly to issue a Rest based call and parse results using SQLXML.

Limitations

- "application/xml" mime type in both "Accept" and "Content-Type" is currently not supported
- File, Map proeprties are currently not supported, thus any multi-part payloads are not supported
- Security metadata is currently not supported
- Custom properties that start with "x-" are not supported.
- Schema with "allof", "multipleof", "items" from JSON schema are not supported

OLAP Translator

The OLAP Services translator, known by the type name *olap*, exposes stored procedures for calling analysis sevices backed by a OLAP server using MDX query language. This translator exposes a stored procedure, invokeMDX, that returns a result set containing tuple array values for a given MDX query. invokeMDX will commonly be used with the ARRAYTABLE table function to extract the results.

Since the Cube metadata exposed by the OLAP servers and relational database metadata are so different, there is no single way to map the metadata from one to other. It is best to query OLAP system using its own native MDX language through. MDX queries my be defined statically or built dynamically in Teiid's abstraction layers.

Usage

The olap translator exposes one low level procedure for accessing olap services.

InvokeMDX Procedure

`invokeMdx` returns a resultset of the tuples as array values.

```
Procedure invokeMdx(mdx in STRING, params VARIADIC OBJECT) returns table (tuple object)
```

The mdx parameter is a MDX query to be executed on the OLAP server.

The results of the query will be returned such that each row on the row axis will be packed into an array value that will first contain each hierarchy member name on the row axis then each measure value from the column axis.

The use of Data Roles should be considered to prevent arbitrary MDX from being submitted to the invokeMDX procedure.

Native Queries

OLAP source procedures may be created using the teiid_rel:native-query extension - see Parameterizable Native Queries.

The parameter value substitution directly inserts boolean, and number values, and treats all other values as string literals.

The procedure will invoke the native-query similar to an invokeMdx call with the benefits that the query is predetermined and that result column types are known, rather than requiring the use of ARRAYTABLE or similar functionality.

Direct Query Procedure

The invokeMdx procedure is the direct query procedure for the OLAP translator. It may be disabled or have it's name changed via the common direct query translator properties just like any other source. A call to the direct query procedure without any parameters will not attempt to parse the mdx query for parameterization. If parameters are used, the value substitution directly inserts boolean, and number values, and treats all other values as string literals.

JCA Resource Adapter

The resource adapter for this translator provided through data source in WildFly, Refer to Admin Guide for "JDBC Data Sources" configuration section. Two sample xml files are provided for accessing OLAP servers in the teiid-examples section. One is Mondrian specific, when Mondrian server is deployed in the same WildFly as Teiid (mondrian-ds.xml). To access any other OLAP servers using XMLA interface, the data source for them can be created using them example template olap-xmla-ds.xml

Note	Due to a classloading change with Mondrian 3.6 and later, a workaround is needed to use a later driver - TEIID-4617 The olap translator module.xml under modules/system/layers/dv/org/jboss/teiid/translator/olap/main/ needs to have a dependency to the Mondrian driver module.

Salesforce Translators

The Salesforce translator supports the SELECT, DELETE, INSERT, UPSERT, and UPDATE operations against a Salesforce.com account. It is designed for use with the Teiid Salesforce resource adapter.

Salesforce API Version Support

salesforce

The translator, known by the type name ***salesforce***, provides Salesforce API 22.0 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 deprecated.

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.

Other API Versions

If you need connectivity to an API version other than what is built in, please utilize the project https://github.com/teiid/salesforce to generate new resource adapter / translator pair.

Execution Properties

Name	Description	Default
ModelAuditFeilds	Audit Model Fields	false
MaxBulkInsertBatchSize	Batch Size to use to insert bulk inserts	2048
SupportsGroupBy	Supports Group By Pushdown. Set to false to have Teiid process group by aggregations, such as those returning more than 2000 rows which error in SOQL.	true

The Salesforce translator can import metadata.

Import Properties

Property Name	Description	Required	Default
NormalizeNames	If the importer should attempt to modify the object/field names so that they can be used unquoted.	false	true

excludeTables	A case-insensitive regular expression that when matched against a table name will exclude it from import. Applied after table names are retrieved. Use a negative look-ahead (?!<inclusion pattern>).* to act as an inclusion filter.	false	n/a
includeTables	A case-insensitive regular expression that when matched against a table name will be included during import. Applied after table names are retrieved from source.	false	n/a
importStatstics	Retrieves cardinalities during import using the REST API explain plan feature.	false	false

NOTE When both *includeTables* and *excludeTables patterns are present during the import, the _includeTables* pattern matched first, then the *excludePatterns* will be applied.

TIP Teiid translator uses Salesforce Partner API libraries to access the Salesforce system. By default Teiid uses Partner API version "22". Metadata retrieval using the translator and execution of queries against Salesforce using this metadata are **strongly** dependent upon the API version that has been used. It is required that you use same API versions for both Metadata retrieval and query execution. For this reason if you have older VDBS then you either need to re-import the metadata or keep using the default version. For any reason you need to use a different version of API than "22", then you can provide the new translator and resource-adapter combination for that API. For your convenience we provided a template project https://github.com/teiid/salesforce that you can build and deploy into Teiid system for the new version of the API. If you are using Teiid Embedded, then you could simply choose to replace the kitted salesforce jars with a later version. Up to version v34 is known to be compatible.

Extension Metadata Properties

Salesforce is not relational database, however Teiid provides ways to map Saleforce data into relational constructs like Tables and Procedures. You can define a foreign Table using DDL in Teiid VDB, which maps to Salesforce's SObject. At runtime, to interpret this table back to a SObject, Teiid decorates or tags this table definition with additional metadata. For example, a table is defined as

```
CREATE FOREIGN TABLE Pricebook2 (
        Id string,
        Name string,
        IsActive boolean,
        IsStandard boolean,
        Description string,
        IsDeleted boolean)
        OPTIONS (
          UPDATABLE 'TRUE',
          "teiid_sf:Supports Query" 'TRUE');
```

In the above the property in OPTIONS clause with property "teiid_sf:Supports Query" annotating that this tables supports SELECT commands. The below are list of metadata extension properties that can be used on Salesforce schema.

Property Name	Description	Required	Default	Applies To

Supports Query	The table supports SELECT commands	false	true	Table
Supports Retrieve	The table supports retrieval of results as result of SELECT commands	false	true	Table

SQL Processing

Salesforce does not provide the same set of functionality as a relational database. For example, Salesforce does not support arbitrary joins between tables. However, working in combination with the Teiid Query Planner, the Salesforce connector supports nearly all of the SQL syntax supported by the Teiid.

The Salesforce Connector executes SQL commands by "pushing down" the command to Salesforce whenever possible, based on the supported capabilities. Teiid will automatically provide additional database functionality when the Salesforce Connector does not explicitly provide support for a given SQL construct. In cases where certain SQL capabilities cannot be pushed down to Salesforce, Teiid will push down the capabilities that are supported, and fetch a set of data from Salesforce. Then, Teiid will evaluate the additional capabilities, creating a subset of the original data set. Finally, Teiid will pass the result to the client.

If you are issuing queries with a group by clause and receive an error for salesforce related to queryMore not being supported, you may either add limits or set the execution property SupportsGroupBy to false.

```
SELECT array_agg(Reports) FROM Supervisor where Division = 'customer support';
```

Neither Salesforce nor the Salesforce Connector support the array_agg() scalar, but they do support CompareCriteriaEquals, so the query that is passed to Salesforce by the connector will be transformed to this query.

```
SELECT Reports FROM Supervisor where Division = 'customer support';
```

The array_agg() function will be applied by the Teiid Query Engine to the result set returned by the connector.

In some cases multiple calls to the Salesforce application will be made to support the SQL passed to the connector.

```
DELETE From Case WHERE Status = 'Closed';
```

The API in Salesforce to delete objects only supports deleting by ID. In order to accomplish this the Salesforce connector will first execute a query to get the IDs of the correct objects, and then delete those objects. So the above DELETE command will result in the following two commands.

```
SELECT ID From Case WHERE Status = 'Closed';
DELETE From Case where ID IN (<result of query>);
```

NOTE The Salesforce API DELETE call is not expressed in SQL, but the above is an equivalent SQL expression.

It’s useful to be aware of unsupported capabilities, in order to avoid fetching large data sets from Salesforce and making you queries as performant as possible. See all Supported Capabilities.

Selecting from Multi-Select Picklists

A multi-select picklist is a field type in Salesforce that can contain multiple values in a single field. Query criteria operators for fields of this type in SOQL are limited to EQ, NE, includes and excludes. The full Salesforce documentation for selecting from mullti-select picklists can be found at the following link Querying Mulit-select Picklists

Teiid SQL does not support the includes or excludes operators, but the Salesforce connector provides user defined function definitions for these operators that provided equivalent functionality for fields of type multi-select. The definition for the functions is:

```
boolean includes(Column column, String param)
boolean excludes(Column column, String param)
```

For example, take a single multi-select picklist column called Status that contains all of these values.

- current
- working
- critical

For that column, all of the below are valid queries:

```
SELECT * FROM Issue WHERE true = includes (Status, 'current, working' );
SELECT * FROM Issue WHERE true = excludes (Status, 'current, working' );
SELECT * FROM Issue WHERE true = includes (Status, 'current;working, critical' );
```

EQ and NE criteria will pass to Salesforce as supplied. For example, these queries will not be modified by the connector.

```
SELECT * FROM Issue WHERE Status = 'current';
SELECT * FROM Issue WHERE Status = 'current;critical';
SELECT * FROM Issue WHERE Status != 'current;working';
```

Selecting All Objects

The Salesforce connector supports the calling the queryAll operation from the Salesforce API. The queryAll operation is equivalent to the query operation with the exception that it returns data about all current and deletedobjects in the system.

The connector determines if it will call the query or queryAll operation via reference to the isDeleted property present on each Salesforce object, and modeled as a column on each table generated by the importer. By default this value is set to False when the model is generated and thus the connector calls query. Users are free to change the value in the model to True, changing the default behaviour of the connector to be queryAll.

The behavior is different if isDeleted is used as a parameter in the query. If the isDeleted column is used as a parameter in the query, and the value is 'true' the connector will call queryAll.

```
select * from Contact where isDeleted = true;
```

If the isDeleted column is used as a parameter in the query, and the value is 'false' the connector perform the default behavior will call query.

```
select * from Contact where isDeleted = false;
```

Selecting Updated Objects

If the option is selected when importing metadata from Salesforce, a GetUpdated procedure is generated in the model with the following structure:

```
GetUpdated (ObjectName IN string,
    StartDate IN datetime,
    EndDate IN datetime,
    LatestDateCovered OUT datetime)
returns
    ID string
```

See the description of the GetUpdated operation in the Salesforce documentation for usage details.

Selecting Deleted Objects

If the option is selected when importing metadata from Salesforce, a GetDeleted procedure is generated in the model with the following structure:

```
GetDeleted (ObjectName IN string,
    StartDate IN datetime,
    EndDate IN datetime,
    EarliestDateAvailable OUT datetime,
    LatestDateCovered OUT datetime)
returns
    ID string,
    DeletedDate datetime
```

See the description of the GetDeleted operation in the Salesforce documentation for usage details.

Relationship Queries

Salesforce does not support joins like a relational database, but it does have support for queries that include parent-to-child or child-to-parent relationships between objects. These are termed Relationship Queries. The SalesForce connector supports Relationship Queries through Outer Join syntax.

```
SELECT Account.name, Contact.Name from Contact LEFT OUTER JOIN Account
on Contact.Accountid = Account.id
```

This query shows the correct syntax to query a SalesForce model with to produce a relationship query from child to parent. It resolves to the following query to SalesForce.

```
SELECT Contact.Account.Name, Contact.Name FROM Contact
```

```
select Contact.Name, Account.Name from Account Left outer Join Contact
on Contact.Accountid = Account.id
```

This query shows the correct syntax to query a SalesForce model with to produce a relationship query from parent to child. It resolves to the following query to SalesForce.

```
SELECT Account.Name, (SELECT Contact.Name FROM
Account.Contacts) FROM Account
```

See the description of the Relationship Queries operation in the SalesForce documentation for limitations.

Bulk Insert Queries

SalesForce translator also supports bulk insert statements using JDBC batch semantics or SELECT INTO semantics. The batch size is determined by the execution property *MaxBulkInsertBatchSize*, which can be overridden in the vdb.xml file. The default value of the batch is 2048. The bulk insert feature uses the async REST based API exposed by Salesforce for execution for better performance.

Bulk Selects

When quering large tables (typically over 10,000,000 records) or if experiencing timeouts with just result batching, Teiid can issue queries to Salesforce using the bulk API. When using a bulk select, PK chunking will be enabled if supported by the query.

The use of the bulk api requires a source hint in the query:

```
SELECT /*+ sh salesforce:'bulk' */ Name ... FROM Account
```

Where salesforce is the source name of the target source.

The default chunk size of 100,000 records will be used.

Note: this feature is only supported by Salsforce API equal to greater than 28, since the default "salesforce" translator uses version 22 of API, it recommended to use "salesforce-34" version of the translator to use this feature.

Supported Capabilities

The following are the capabilities supported by the Salesforce Connector. These SQL constructs will be pushed down to Salesforce.

- SELECT command
- INSERT Command
- UPDATE Command
- DELETE Command
- NotCriteria
- OrCriteria
- CompareCriteriaEquals
- CompareCriteriaOrdered
- IsNullCritiera
- InCriteria
- LikeCriteria - Supported for String fields only.
- RowLimit
- Basic Aggregates
- OuterJoins with join criteria KEY

Native Queries

Salesforce procedures may optionally have native queries associated with them - see Parameterizable Native Queries. The operation prefix (select;, insert;, update;, delete; - see below for more) must be present in the native-query, but it will not be issued as part of the query to the source.

Example DDL for a SF native procedure

```
CREATE FOREIGN PROCEDURE proc (arg1 integer, arg2 string) OPTIONS ("teiid_rel:native-query" 'search;SELECT ...
complex SOQL ... WHERE col1 = $1 and col2 = $2') returns (col1 string, col2 string, col3 timestamp);
```

Direct Query Procedure

This feature is turned off by default because of the security risk this exposes to execute any command against the source. To enable this feature, override the execution property called _SupportsDirectQueryProcedure to true.

Tip	By default the name of the procedure that executes the queries directly is **native**. Override the execution property _DirectQueryProcedureName to change it to another name.

The Salesforce translator provides a procedure to execute any ad-hoc SOQL query directly against the source without Teiid parsing or resolving. Since the metadata of this procedure's results are not known to Teiid, they are returned as an object array. ARRAYTABLE can be used construct tabular output for consumption by client applications. Teiid exposes this procedure with a simple query structure as follows:

Select

Select Example

```
SELECT x.* FROM (call sf_source.native('search;SELECT Account.Id, Account.Type, Account.Name FROM Account')) w,
 ARRAYTABLE(w.tuple COLUMNS "id" string , "type" string, "name" String) AS x
```

from the above code, the "search" keyword followed by a query statement.

Note	The SOQL is treated as a parameterized native query so that parameter values may be inserted in the query string properly - see Parameterizable Native Queries

The results returned by search may contain the object Id as the first column value regardless of whether it was selected. Also queries that select columns from multiple object types will not be correct.

Delete

Delete Example

```
SELECT x.* FROM (call sf_source.native('delete;', 'id1', 'id2')) w,
 ARRAYTABLE(w.tuple COLUMNS "updatecount" integer) AS x
```

form the above code, the "delete;" keyword followed by the ids to delete as varargs.

Create or Update

Create Example

```
SELECT x.* FROM
 (call sf_source.native('create;type=table;attributes=one,two,three', 'one', 2, 3.0)) w,
 ARRAYTABLE(w.tuple COLUMNS "update_count" integer) AS x
```

form the above code, the "create" or "update" keyword must be followed by the following properties. Attributes must be matched positionally by the procedure variables - thus in the example attribute two will be set to 2.

Property Name	Description	Required
type	Table Name	Yes
attributes	comma separated list of names of the columns	no

The values for each attribute is specified as separate argument to the "native" procedure.

Update is similar to create, with one more extra property called "id", which defines identifier for the record.

Update Example

```
SELECT x.* FROM
 (call sf_source.native('update;id=pk;type=table;attributes=one,two,three', 'one', 2, 3.0)) w,
 ARRAYTABLE(w.tuple COLUMNS "update_count" integer) AS x
```

Tip	By default the name of the procedure that executes the queries directly is called native, however user can + set override execution property vdb.xml file to change it.

JCA Resource Adapter

The resource adapter for this translator is provided through Salesforce Data Sources. Refer to Admin Guide for configuration.

SAP Gateway Translator

The SAP Gateway Translator, known by the type name *sap-gateway*, provides a translator for accessing the SAP Gateway using the OData protocol. This translator is extension of OData Translator and uses Teiid WS resource adapter for making web service calls. This translator understands the most of the SAP specific OData extensions to the metadata defined in the document SAP Annotations for OData Version 2.0

When the metadata is imported from SAP Gateway, the Teiid models are created to accordingly for SAP specific *EntitySet* and *Property* annotations defined in document above.

The following "execution properties" are supported in this translator

Execution Properties

Name	Description	Default
DatabaseTimeZone	The time zone of the database. Used when fetchings date, time, or timestamp values	The system default time zone
SupportsOdataCount	Supports $count	true
SupportsOdataFilter	Supports $filter	true
SupportsOdataOrderBy	Supports $orderby	true
SupportsOdataSkip	Supports $skip	true
SupportsOdataTop	Supports $top	true

Based on how you implemented your SAP Gateway service, if can choose to turn off some of the features above.

Note	**Using pagable, topable metadata extensions?** - If metadata on your service defined "pagable" and/or "topable" as "false" on **any** table, you **must** turn off "SupportsOdataTop" and "SupportsOdataSkip" execution-properties in your translator, so that you will not end up with wrong results. SAP metadata has capability to control these in a fine grained fashion any on EnitySet, however Teiid can only control these at translator level.

Note	**SAP Examples** - Sample examples defined at http://scn.sap.com/docs/DOC-31221, we found to be lacking in full metadata in certain examples. For example, "filterable" clause never defined on some properties, but if you send a request $filter it will silently ignore it. You can verify this behavior by directly executing the REST service using a web browser with respective query. So, Make sure you have implemented your service correctly, or you can turn off certain features in this translator by using "execution properties" override. See an example in OData Translator

Web Services Translator

The Web Services translator, known by the type name *ws*, exposes stored procedures for calling web services backed by a Teiid WS resource adapter. The WS resource adapter may optionally be configured to point at a specific WSDL. Results from this translator will commonly be used with the TEXTTABLE or XMLTABLE table functions to use CSV or XML formated data.

Execution Properties

Name	Description	When Used	Default
DefaultBinding	The binding that should be used if one is not specified. Can be one of HTTP, SOAP11, or SOAP12	invoke*	SOAP12
DefaultServiceMode	The default service mode. For SOAP, MESSAGE mode indicates that the request will contain the entire SOAP envelope and not just the contents of the SOAP body. Can be one of MESSAGE or PAYLOAD	invoke* or WSDL call	PAYLOAD
XMLParamName	Used with the HTTP binding (typically with the GET method) to indicate that the request document should be part of the query string.	invoke*	null - unused

Note	Setting the proper binding value on the translator is recommended as it removes the need for callers to pass an explicit value. If your service is actually uses SOAP11, but the binding used SOAP12 you will receive execution failures.

There are no ws importer settings, but it can provide metadata for VDBs. If the connection is configured to point at a specific WSDL, the translator will import all SOAP operations under the specified service and port as procedures.

Importer Properties

When specifying the importer property, it must be prefixed with "importer.". Example: importer.tableTypes

Name	Description	Default
importWSDL	Import the metadata from the WSDL URL configured in resource-adapter	true

Usage

The WS translator exposes low level procedures for accessing web services. See also the twitter example in the kit.

Invoke Procedure

Invoke allows for multiple binding, or protocol modes, including HTTP, SOAP11, and SOAP12.

```
Procedure invoke(binding in STRING, action in STRING, request in XML, endpoint in STRING, stream in BOOLEAN) re
turns XML
```

The binding may be one of null (to use the default) HTTP, SOAP11, or SOAP12. Action with a SOAP binding indicates the SOAPAction value. Action with a HTTP binding indicates the HTTP method (GET, POST, etc.), which defaults to POST.

A null value for the binding or endpoint will use the default value. The default endpoint is specified in the WS resource adapter configuration. The endpoint URL may be absolute or relative. If it's relative then it will be combined with the default endpoint.

Since multiple parameters are not required to have values, it is often more clear to call the invoke procedure with named parameter syntax.

```
call invoke(binding=>'HTTP', action=>'GET')
```

The request XML should be a valid XML document or root element.

InvokeHTTP Procedure

`invokeHttp` can return the byte contents of an HTTP(S) call.

```
Procedure invokeHttp(action in STRING, request in OBJECT, endpoint in STRING, stream in BOOLEAN, contentType ou
t STRING, headers in CLOB) returns BLOB
```

Action indicates the HTTP method (GET, POST, etc.), which defaults to POST.

A null value for endpoint will use the default value. The default endpoint is specified in the WS resource adapter configuration. The endpoint URL may be absolute or relative. If it's relative then it will be combined with the default endpoint.

Since multiple parameters are not required to have values, it is often more clear to call the invoke procedure with named parameter syntax.

```
call invokeHttp(action=>'GET')
```

The request can be one of SQLXML, STRING, BLOB, or CLOB. The request will be sent as the POST payload in byte form. For STRING/CLOB values this will default to the UTF-8 encoding. To control the byte encoding, see the to_bytes function.

The optional headers parameter can be used to specify the request header values as a JSON value. The JSON value should be a JSON object with primitive or list of primitive values.

```
call invokeHttp(... headers=>jsonObject('application/json' as "Content-Type", jsonArray('gzip', 'deflate') as "
Accept-Encoding"))
```

Recommendations for setting headers parameter:

- Content-Type may be necessary if HTTP POST/PUT method is invoked
- *Accept* is necessary if you want to control return Media Type

WSDL based Procedures

The procedures above give you anonymous way to execute any web service methods by supplying an endpoint, with this mechanism you can alter the endpoint defined in WSDL with a different endpoint. However, if you have access to the WSDL, then you can configure the WSDL URL in the web-service resource-adapter's connection configuration, Web Service translator can parse the WSDL and provide the methods under configured port as pre-built procedures as its metadata. If you are using the default native metadata import, you will see the procedures in your web service's source model.

Note	**Native queries** - Native queries or a direct query execution procedure is not supported on the Web Services Translator.

Streaming Considerations

If the stream parameter is set to true, then the resulting lob value may only be used a single time. If stream is null or false, then the engine may need to save a copy of the result for repeated use. Care must be used as some operations, such as casting or XMLPARSE may perform validation which results in the stream being consumed.

JCA Resource Adapter

The resource adapter for this translator is a Web Service Data Source.

Note	**WS-Security** - Currently you can only use WSDL based Procedures participate in WS-Security, when resource-adapter is configured with correct CXF configuration.

Federated Planning

Teiid at its core is a federated relational query engine. This query engine allows you to treat all of your data sources as one virtual database and access them in a single SQL query. This allows you to focus on building your application, not on hand-coding joins, and other relational operations, between data sources.

Child Pages

- Planning Overview
- Query Planner
- Query Plans
- Federated Optimizations
- Subquery Optimization
- XQuery Optimization
- Federated Failure Modes
- Conformed Tables

Planning Overview

When the query engine receives an incoming SQL query it performs the following operations:

1. **Parsing** - validate syntax and convert to internal form
2. **Resolving** - link all identifiers to metadata and functions to the function library
3. **Validating** - validate SQL semantics based on metadata references and type signatures
4. **Rewriting** - rewrite SQL to simplify expressions and criteria
5. **Logical plan optimization** - the rewritten canonical SQL is converted into a logical plan for in-depth optimization. The Teiid optimizer is predominantly rule-based. Based upon the query structure and hints a certain rule set will be applied. These rules may trigger in turn trigger the execution of more rules. Within several rules, Teiid also takes advantage of costing information. The logical plan optimization steps can be seen by using SET SHOWPLAN DEBUG clause, a sample steps are described in Query Planner#Reading a Debug Plan. More details about logical plan's nodes and rule-based optimization refer to Query Planner.
6. **Processing plan conversion** - the logic plan is converted into an executable form where the nodes are representative of basic processing operations. The final processing plan is displayed as the Query Plans.

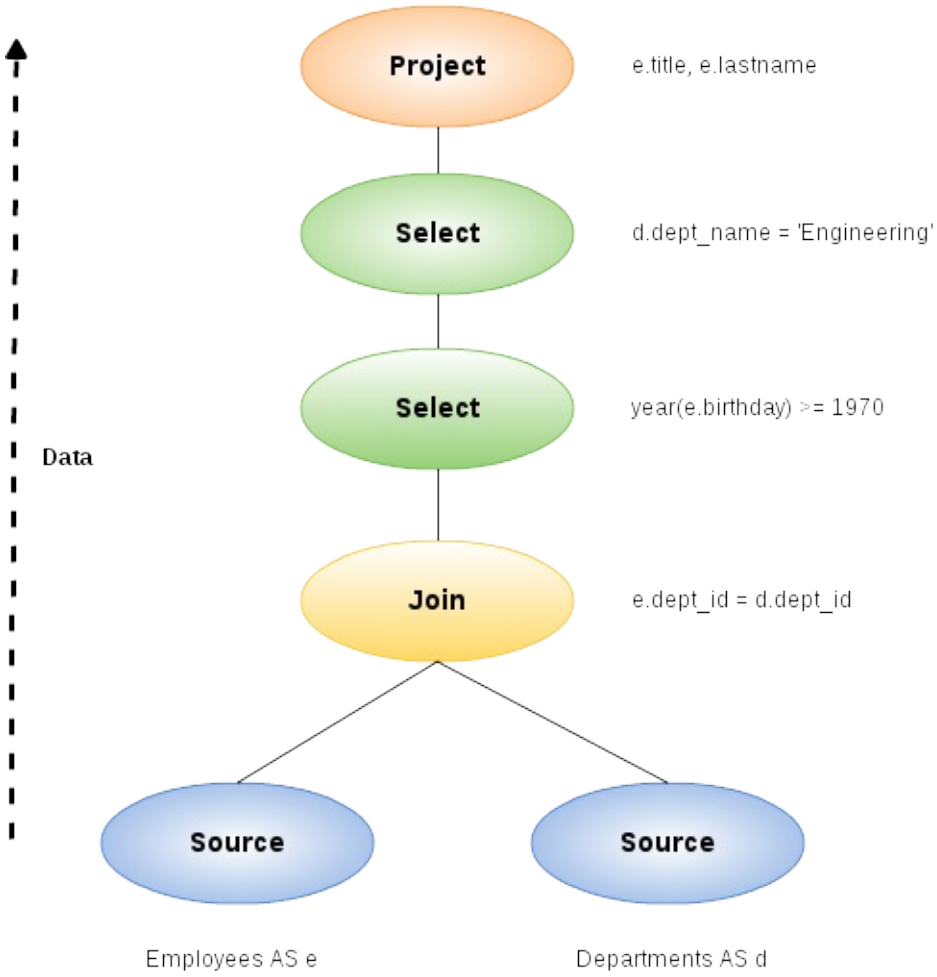
The logical query plan is a tree of operations used to transform data in source tables to the expected result set. In the tree, data flows from the bottom (tables) to the top (output). The primary logical operations are *select* (select or filter rows based on a criteria), *project* (project or compute column values), *join* , *source* (retrieve data from a table), *sort* (ORDER BY), *duplicate removal* (SELECT DISTINCT), *group* (GROUP BY), and *union* (UNION).

For example, consider the following query that retrieves all engineering employees born since 1970.

Example query

```
SELECT e.title, e.lastname FROM Employees AS e JOIN Departments AS d ON e.dept_id = d.dept_id WHERE year(e.birt
hday) >= 1970 AND d.dept_name = 'Engineering'
```

Logically, the data from the Employees and Departments tables are retrieved, then joined, then filtered as specified, and finally the output columns are projected. The canonical query plan thus looks like this:



Data flows from the tables at the bottom upwards through the join, through the select, and finally through the project to produce the final results. The data passed between each node is logically a result set with columns and rows.

Of course, this is what happens **logically**, not how the plan is actually executed. Starting from this initial plan, the query planner performs transformations on the query plan tree to produce an equivalent plan that retrieves the same results faster. Both a federated query planner and a relational database planner deal with the same concepts and many of the same plan transformations. In this example, the criteria on the Departments and Employees tables will be pushed down the tree to filter the results as early as possible.

In both cases, the goal is to retrieve the query results in the fastest possible time. However, the relational database planner does this primarily by optimizing the access paths in pulling data from storage.

In contrast, a federated query planner is less concerned about storage access because it is typically pushing that burden to the data source. The most important consideration for a federated query planner is minimizing data transfer.

Query Planner

- Canonical Plan and All Nodes
- Node Properties
  - Access Properties
  - Set operation Properties
  - Join Properties
  - Project Properties
  - Select Properties
  - Sort Properties
  - Source Properties
  - Group Properties
  - Tuple Limit Properties
  - General and Costing Properties
- Rules

For each sub-command in the user command an appropriate kind of sub-planner is used (relational, XML, procedure, etc).

Each planner has three primary phases:

1. Generate canonical plan
2. Optimization
3. Plan to process converter - converts plan data structure into a processing form

Relational Planner

A relational processing plan is created by the optimizer after the logical plan is manipulated by a series of rules. The application of rules is determined both by the query structure and by the rules themselves. The node structure of the debug plan resembles that of the processing plan, but the node types more logically represent SQL operations.

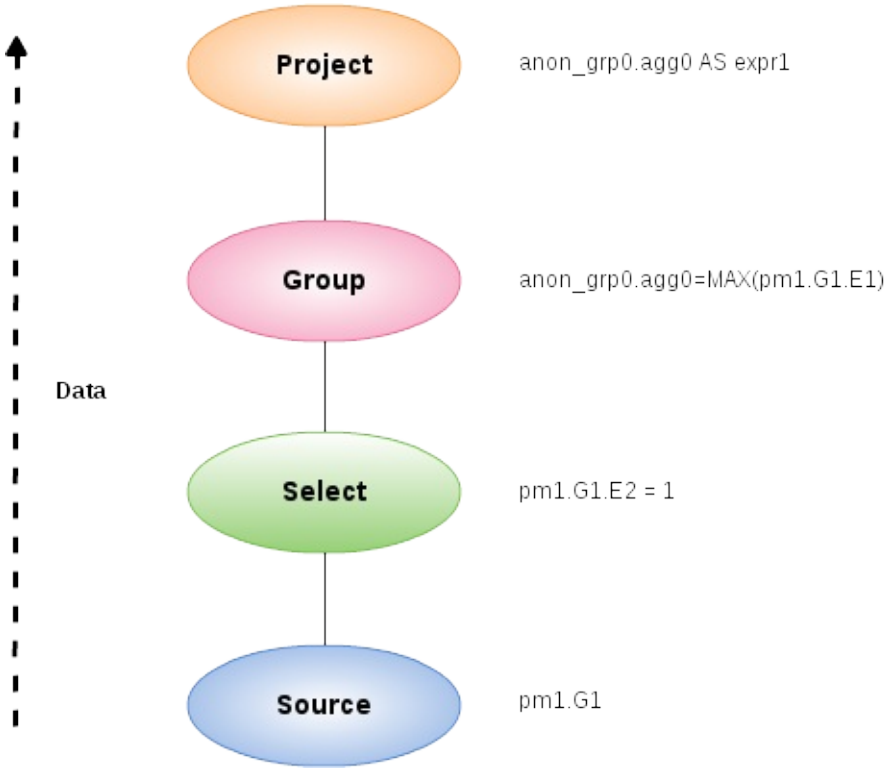
Canonical Plan and All Nodes

As Planning Overview, a user SQL statement after Parsing, Resolving, Validating, Rewriting, it be converted into a canonical plan form. The canonical plan form most closely resembles the initial SQL structure. A SQL select query has the following possible clauses (all but SELECT are optional): WITH, SELECT, FROM, WHERE, GROUP BY, HAVING, ORDER BY, LIMIT. These clauses are logically executed in the following order:

1. WITH (create common table expressions) - handled by a specialized PROJECT NODE
2. FROM (read and join all data from tables) - SOURCE node for each from clause item, Join node (if >1 table)
3. WHERE (filter rows) - SELECT node
4. GROUP BY (group rows into collapsed rows) - GROUP node

5. HAVING (filter grouped rows) - SELECT node
6. SELECT (evaluate expressions and return only requested rows) - PROJECT node and DUP_REMOVE node (for SELECT DISTINCT)
7. INTO - specialized PROJECT with a SOURCE child
8. ORDER BY (sort rows) - SORT node
9. LIMIT (limit result set to a certain range of results) - LIMIT node

For example, a SQL statement such as `SELECT max(pm1.g1.e1) FROM pm1.g1 WHERE e2 = 1` creates a logical plan:




```
Project(groups=[anon_grp0], props={PROJECT_COLS=[anon_grp0.agg0 AS expr1]})
  Group(groups=[anon_grp0], props={SYMBOL_MAP={anon_grp0.agg0=MAX(pm1.G1.E1)}})
    Select(groups=[pm1.G1], props={SELECT_CRITERIA=pm1.G1.E2 = 1})
      Source(groups=[pm1.G1])
```

Here the Source corresponds to the FROM clause, the Select corresponds to the WHERE clause, the Group corresponds to the implied grouping to create the max aggregate, and the Project corresponds to the SELECT clause.

Note	The affect of grouping generates what is effectively an inline view, anon_grp0, to handle the projection of values created by the grouping.

Table 1. **Node Types**

Type Name	Description
ACCESS	a source access or plan execution.
DUP_REMOVE	removes duplicate rows
JOIN	a join (LEFT OUTER, FULL OUTER, INNER, CROSS, SEMI, etc.)

PROJECT	a projection of tuple values
SELECT	a filtering of tuples
SORT	an ordering operation, which may be inserted to process other operations such as joins
SOURCE	any logical source of tuples including an inline view, a source access, XMLTABLE, etc.
GROUP	a grouping operation
SET_OP	a set operation (UNION/INTERSECT/EXCEPT)
NULL	a source of no tuples
TUPLE_LIMIT	row offset / limit

Node Properties

Each node has a set of applicable properties that are typically shown on the node.

Access Properties

Table 2. **Access Properties**

Property Name	Description
ATOMIC_REQUEST	The final form of a source request
MODEL_ID	The metadata object for the target model/schema
PROCEDURE_CRITERIA/PROCEDURE_INPUTS/PROCEDURE_DEFAULTS	Used in planning procedureal relational queries
IS_MULTI_SOURCE	set to true when the node represents a multi-source access
SOURCE_NAME	used to track the multi-source source name
CONFORMED_SOURCES	tracks the set of conformed sources when the conformed extension metadata is used
SUB_PLAN/SUB_PLANS	used in multi-source planning

Set operation Properties

Table 3. **Set operation Properties**

Property Name	Description
SET_OPERATION/USE_ALL	defines the set operation(UNION/INTERSECT/EXCEPT) and if all rows or distinct rows are used.

Join Properties

Table 4. **Join Properties**

Property Name	Description
JOIN_CRITERIA	all join predicates
JOIN_TYPE	type of join (INNER, LEFT OUTER, etc.)
JOIN_STRATEGY	the algorithm to use (nested loop, merge, etc.)
LEFT_EXPRESSIONS	the expressions in equi-join predicates that originate from the left side of the join
RIGHT_EXPRESSIONS	the expressions in equi-join predicates that originate from the right side of the join
DEPENDENT_VALUE_SOURCE	set if a dependent join is used
NON_EQUI_JOIN_CRITERIA	non-equi join predicates
SORT_LEFT	if the left side needs sorted for join processing
SORT_RIGHT	if the right side needs sorted for join processing
IS_OPTIONAL	if the join is optional
IS_LEFT_DISTINCT	if the left side is distinct with respect to the equi join predicates
IS_RIGHT_DISTINCT	if the right side is distinct with respect to the equi join predicates
IS_SEMI_DEP	if the dependent join represents a semi-join
PRESERVE	if the preserve hint is preserving the join order

Project Properties

Table 5. **Project Properties**

Property Name	Description
PROJECT_COLS	the expressions projected
INTO_GROUP	the group targeted if this is a select into or insert with a query expression
HAS_WINDOW_FUNCTIONS	true if window functions are used
CONSTRAINT	the constraint that must be met if the values are being projected into a group
UPSERT	If the insert is an upsert

Select Properties

Table 6. **Select Properties**

Property Name	Description
SELECT_CRITERIA	the filter
IS_HAVING	if the filter is applied after grouping
IS_PHANTOM	true if the node is marked for removal, but temporarily left in the plan.
IS_TEMPORARY	inferred criteria that may not be used in the final plan
IS_COPIED	if the criteria has already been processed by rule copy criteria
IS_PUSHED	if the criteria is pushed as far as possible
IS_DEPENDENT_SET	if the criteria is the filter of a dependent join

Sort Properties

Table 7. **Sort Properties**

Property Name	Description
SORT_ORDER	the order by that defines the sort
UNRELATED_SORT	if the ordering includes a value that is not being projected
IS_DUP_REMOVAL	if the sort should also perform duplicate removal over the entire projection

Source Properties

Table 8. **Source Properties**

Property Name	Description
SYMBOL_MAP	the mapping from the columns above the source to the projected expressions. Also present on Group nodes
PARTITION_INFO	the partitioning of the union branches
VIRTUAL_COMMAND	if the source represents an view or inline view, the query that defined the view
MAKE_DEP	hint information
PROCESSOR_PLAN	the processor plan of a non-relational source(typically from the NESTED_COMMAND)
NESTED_COMMAND	the non-relational command

TABLE_FUNCTION	the table function (XMLTABLE, OBJECTTABLE, etc.) defining the source
CORRELATED_REFERENCES	the correlated references for the nodes below the source
MAKE_NOT_DEP	if make not dep is set
INLINE_VIEW	If the source node represents an inline view
NO_UNNEST	if the no_unnest hint is set
MAKE_IND	if the make ind hint is set
SOURCE_HINT	the source hint. See Federated Optimizations.
ACCESS_PATTERNS	access patterns yet to be satisfied
ACCESS_PATTERN_USED	satisfied access patterns
REQUIRED_ACCESS_PATTERN_GROUPS	groups needed to satisfy the access patterns. Used in join planning.

Note	Many source properties also become present on associated access nodes.

Group Properties

Table 9. **Group Properties**

Property Name	Description
GROUP_COLS	the grouping columns
ROLLUP	if the grouping includes a rollup

Tuple Limit Properties

Table 10. **Tuple Limit Properties**

Property Name	Description
MAX_TUPLE_LIMIT	expression that evaluates to the max number of tuples generated
OFFSET_TUPLE_COUNT	Expression that evaluates to the tuple offset of the starting tuple
IS_IMPLICIT_LIMIT	if the limit is created by the rewriter as part of a subquery optimization
IS_NON_STRICT	if the unordered limit should not be enforced strictly

General and Costing Properties

Table 11. **General and Costing Properties**

Property Name	Description
OUTPUT_COLS	the output columns for the node. Is typically set after rule assign output elements.
EST_SET_SIZE	represents the estimated set size this node would produce for a sibling node as the independent node in a dependent join scenario
EST_DEP_CARDINALITY	value that represents the estimated cardinality (amount of rows) produced by this node as the dependent node in a dependent join scenario
EST_DEP_JOIN_COST	value that represents the estimated cost of a dependent join (the join strategy for this could be Nested Loop or Merge)
EST_JOIN_COST	value that represents the estimated cost of a merge join (the join strategy for this could be Nested Loop or Merge)
EST_CARDINALITY	represents the estimated cardinality (amount of rows) produced by this node
EST_COL_STATS	column statistics including number of null values, distinct value count, etc.
EST_SELECTIVITY	represents the selectivity of a criteria node

Rules

Relational optimization is based upon rule execution that evolves the initial plan into the execution plan. There are a set of pre-defined rules that are dynamically assembled into a rule stack for every query. The rule stack is assembled based on the contents of the user's query and the views/procedures accessed. For example, if there are no view layers, then rule Merge Virtual, which merges view layers together, is not needed and will not be added to the stack. This allows the rule stack to reflect the complexity of the query.

Logically the plan node data structure represents a tree of nodes where the source data comes up from the leaf nodes (typically Access nodes in the final plan), flows up through the tree and produces the user's results out the top. The nodes in the plan structure can have bidirectional links, dynamic properties, and allow any number of child nodes. Processing plans in contrast typically have fixed properties.

Plan rule manipulate the plan tree, fire other rules, and drive the optimization process. Each rule is designed to perform a narrow set of tasks. Some rules can be run multiple times. Some rules require a specific set of precursors to run properly.

- Access Pattern Validation - ensures that all access patterns have been satisfied
- Apply Security - applies row and column level security
- Assign Output Symbol - this rule walks top down through every node and calculates the output columns for each node. Columns that are not needed are dropped at every node, which is known as projection minimization. This is done by keeping track of both the columns needed to feed the parent node and also keeping track of columns that are "created" at a certain node.
- Calculate Cost - adds costing information to the plan
- Choose Dependent - this rule looks at each join node and determines whether the join should be made dependent and in which direction. Cardinality, the number of distinct values, and primary key information are used in several formulas to determine whether a dependent join is likely to be worthwhile. The dependent join differs in performance ideally because a

fewer number of values will be returned from the dependent side. Also, we must consider the number of values passed from independent to dependent side. If that set is larger than the max number of values in an IN criteria on the dependent side, then we must break the query into a set of queries and combine their results. Executing each query in the connector has some overhead and that is taken into account. Without costing information a lot of common cases where the only criteria specified is on a non-unique (but strongly limiting) field are missed. A join is eligible to be dependent if:

- there is at least one equi-join criterion, i.e. tablea.col = tableb.col
- the join is not a full outer join and the dependent side of the join is on the inner side of the join

The join will be made dependent if one of the following conditions, listed in precedence order, holds:

- There is an unsatisfied access pattern that can be satisfied with the dependent join criteria
- The potential dependent side of the join is marked with an option makedep
- (4.3.2) if costing was enabled, the estimated cost for the dependent join (5.0+ possibly in each direction in the case of inner joins) is computed and compared to not performing the dependent join. If the costs were all determined (which requires all relevant table cardinality, column ndv, and possibly nnv values to be populated) the lowest is chosen.
- If key metadata information indicates that the potential dependent side is not "small" and the other side is "not small" or (5.0.1) the potential dependent side is the inner side of a left outer join.

Dependent join is the key optimization we use to efficiently process multi-source joins. Instead of reading all of source A and all of source B and joining them on A.x = B.x, we read all of A then build a set of A.x that are passed as a criteria when querying B. In cases where A is small and B is large, this can drastically reduce the data retrieved from B, thus greatly speeding the overall query.

- Choose Join Strategy - choose the join strategy based upon the cost and attributes of the join.
- Clean Criteria - removes phantom criteria
- Collapse Source - takes all of the nodes below an access node and creates a SQL query representation
- Copy Criteria - this rule copies criteria over an equality criteria that is present in the criteria of a join. Since the equality defines an equivalence, this is a valid way to create a new criteria that may limit results on the other side of the join (especially in the case of a multi-source join).
- Decompose Join - this rule perfomrs a partition-wise join optimization on joins of Federated Optimizations#Partitioned Union. The decision to decompose is based upon detecting that each side of the join is a partitioned union (note that non-ansi joins of more than 2 tables may cause the optimization to not detect the appropriate join). The rule currently only looks for situations where at most 1 partition matches from each side.
- Implement Join Strategy - adds necessary sort and other nodes to process the chosen join strategy
- Merge Criteria - combines select nodes and can convert subqueries to semi-joins
- Merge Virtual - removes view and inline view layers
- Place Access - places access nodes under source nodes. An access node represents the point at which everything below the access node gets pushed to the source or is a plan invocation. Later rules focus on either pushing under the access or pulling the access node up the tree to move more work down to the sources. This rule is also responsible for placing Federated Optimizations#Access Patterns.
- Plan Joins - this rule attempts to find an optimal ordering of the joins performed in the plan, while ensuring that Federated Optimizations#Access Patterns dependencies are met. This rule has three main steps. First it must determine an ordering of joins that satisfy the access patterns present. Second it will heuristically create joins that can be pushed to the source (if a set of joins are pushed to the source, we will not attempt to create an optimal ordering within that set. More than likely it will be

sent to the source in the non-ANSI multi-join syntax and will be optimized by the database). Third it will use costing information to determine the best left-linear ordering of joins performed in the processing engine. This third step will do an exhaustive search for 7 or less join sources and is heuristically driven by join selectivity for 8 or more sources.

- Plan Outer Joins - reorders outer joins as permitted to improve push down.
- Plan Procedures - plans procedures that appear in procedural relational queries
- Plan Sorts - optimizations around sorting, such as combining sort operations or moving projection
- Plan 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 - pushed 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.

One of the most important optimization related to pushing criteria, is how the criteria will be pushed through join. Consider the following plan tree that represents a subtree of the plan for the query `select * from A inner join b on (A.x = B.x) where B.y = 3`

```
SELECT (B.y = 3)
       |
      JOIN - Inner Join on (A.x = B.x)
     /     \
  SRC (A)   SRC (B)
```

Note	SELECT nodes represent criteria, and SRC stands for SOURCE.

It is always valid for inner join and cross joins to push (single source) criteria that are above the join, below the join. This allows for criteria originating in the user query to eventually be present in source queries below the joins. This result can be represented visually as:

```
JOIN - Inner Join on (A.x = B.x)
      /    \
     /   SELECT (B.y = 3)
    |        |
  SRC (A)   SRC (B)
```

The same optimization is valid for criteria specified against the outer side of an outer join. For example:

```
 SELECT (B.y = 3)
       |
      JOIN - Right Outer Join on (A.x = B.x)
     /     \
  SRC (A)   SRC (B)
```

Becomes

```
      JOIN - Right Outer Join on (A.x = B.x)
      /    \
     /   SELECT (B.y = 3)
    |        |
  SRC (A)   SRC (B)
```

However criteria specified against the inner side of an outer join needs special consideration. The above scenario with a left or full outer join is not the same. For example:

```
SELECT (B.y = 3)
     |
    JOIN - Left Outer Join on (A.x = B.x)
   /     \
SRC (A)   SRC (B)
```

Can become (available only after 5.0.2):

```
JOIN - Inner Join on (A.x = B.x)
     /    \
    /   SELECT (B.y = 3)
   |        |
SRC (A)   SRC (B)
```

Since the criterion is not dependent upon the null values that may be populated from the inner side of the join, the criterion is eligible to be pushed below the join – but only if the join type is also changed to an inner join. On the other hand, criteria that are dependent upon the presence of null values CANNOT be moved. For example:

```
SELECT (B.y is null)
     |
    JOIN - Left Outer Join on (A.x = B.x)
   /     \
SRC (A)   SRC (B)
```

This plan tree must have the criteria remain above the join, since the outer join may be introducing null values itself.

- Raise Access - this rule attempts to raise the Access nodes as far up the plan as possible. This is mostly done by looking at the source's capabilities and determining whether the operations can be achieved in the source or not.
- Raise Null - raises null nodes. Raising a null node removes the need to consider any part of the old plan that was below the null node.
- Remove Optional Joins - removes joins that are marked as or determined to be optional
- Substitute Expressions - used only when a function based index is present
- Validate Where All - ensures criteria is used when required by the source

Cost Calculations

The cost of node operations is primarily determined by an estimate of the number of rows (also referred to as cardinality) that will be processed by it. The optimizer will typically compute cardinalities from the bottom up of the plan (or subplan) at several points in time with planning - once generally with rule calculate cost, and then specifically for join planning and other decisions. The cost calculation is mainly directed by the statistics set on physical tables(cardinality, NNV, NDV, etc.) and is also influenced by the presence of constraints (unique, primary key, index, etc.). If there is a situation that seems like a sub-optimal plan is being chosen, you should first ensure that at least representative table cardinalities are set on the physical tables involved.

Reading a Debug Plan

As each relational sub plan is optimized, the plan will show what is being optimized and it's canonical form:

```
OPTIMIZE:
SELECT e1 FROM (SELECT e1 FROM pm1.g1) AS x
```

```
------------------------------------------------------------------------------
GENERATE CANONICAL:
SELECT e1 FROM (SELECT e1 FROM pm1.g1) AS x

CANONICAL PLAN:
Project(groups=[x], props={PROJECT_COLS=[e1]})
  Source(groups=[x], props={NESTED_COMMAND=SELECT e1 FROM pm1.g1, SYMBOL_MAP={x.e1=e1}})
    Project(groups=[pm1.g1], props={PROJECT_COLS=[e1]})
      Source(groups=[pm1.g1])
```

With more complicated user queries, such as a procedure invocation or one containing subqueries, the sub plans may be nested within the overall plan. Each plan ends by showing the final processing plan:

```
------------------------------------------------------------------------------
OPTIMIZATION COMPLETE:
PROCESSOR PLAN:
AccessNode(0) output=[e1] SELECT g_0.e1 FROM pm1.g1 AS g_0
```

The affect of rules can be seen by the state of the plan tree before and after the rule fires. For example, the debug log below shows the application of rule merge virtual, which will remove the "x" inline view layer:

```
EXECUTING AssignOutputElements

AFTER:
Project(groups=[x], props={PROJECT_COLS=[e1], OUTPUT_COLS=[e1]})
  Source(groups=[x], props={NESTED_COMMAND=SELECT e1 FROM pm1.g1, SYMBOL_MAP={x.e1=e1}, OUTPUT_COLS=[e1]})
    Project(groups=[pm1.g1], props={PROJECT_COLS=[e1], OUTPUT_COLS=[e1]})
      Access(groups=[pm1.g1], props={SOURCE_HINT=null, MODEL_ID=Schema name=pm1, nameInSource=null, uuid=3335,
OUTPUT_COLS=[e1]})
        Source(groups=[pm1.g1], props={OUTPUT_COLS=[e1]})


==============================================================================
EXECUTING MergeVirtual

AFTER:
Project(groups=[pm1.g1], props={PROJECT_COLS=[e1], OUTPUT_COLS=[e1]})
  Access(groups=[pm1.g1], props={SOURCE_HINT=null, MODEL_ID=Schema name=pm1, nameInSource=null, uuid=3335, OUTP
UT_COLS=[e1]})
    Source(groups=[pm1.g1])
```

Some important planning decisions are shown in the plan as they occur as an annotation. For example the snippet below shows that the access node could not be raised as the parent select node contained an unsupported subquery.

```
Project(groups=[pm1.g1], props={PROJECT_COLS=[e1], OUTPUT_COLS=null})
  Select(groups=[pm1.g1], props={SELECT_CRITERIA=e1 IN /*+ NO_UNNEST */ (SELECT e1 FROM pm2.g1), OUTPUT_COLS=nu
ll})
    Access(groups=[pm1.g1], props={SOURCE_HINT=null, MODEL_ID=Schema name=pm1, nameInSource=null, uuid=3341, OU
TPUT_COLS=null})
      Source(groups=[pm1.g1], props={OUTPUT_COLS=null})


==============================================================================
EXECUTING RaiseAccess
LOW Relational Planner SubqueryIn is not supported by source pm1 - e1 IN /*+ NO_UNNEST */ (SELECT e1 FROM pm2.g
1) was not pushed

AFTER:
Project(groups=[pm1.g1])
  Select(groups=[pm1.g1], props={SELECT_CRITERIA=e1 IN /*+ NO_UNNEST */ (SELECT e1 FROM pm2.g1), OUTPUT_COLS=nu
ll})
    Access(groups=[pm1.g1], props={SOURCE_HINT=null, MODEL_ID=Schema name=pm1, nameInSource=null, uuid=3341, OU
```

```
TPUT_COLS=null})
      Source(groups=[pm1.g1])
```

Procedure Planner

The procedure planner is fairly simple. It converts the statements in the procedure into instructions in a program that will be run during processing. This is mostly a 1-to-1 mapping and very little optimization is performed.

XML Planner

The XML Planner creates an XML plan that is relatively close to the end result of the Procedure Planner – a program with instructions. Many of the instructions are even similar (while loop, execute SQL, etc). Additional instructions deal with producing the output result document (adding elements and attributes).

The XML planner does several types of planning (not necessarily in this order):

- Document selection - determine which tags of the virtual document should be excluded from the output document. This is done based on a combination of the model (which marks parts of the document excluded) and the query (which may specify a subset of columns to include in the SELECT clause).
- Criteria evaluation - breaks apart the user's criteria, determine which result set the criteria should be applied to, and add that criteria to that result set query.
- Result set ordering - the query's ORDER BY clause is broken up and the ORDER BY is applied to each result set as necessary
- Result set planning - ultimately, each result set is planned using the relational planner and taking into account all the impacts from the user's query. The planner will also look to automatically create staging tables and dependent joins based upon the mapping class hierarchy.
- Program generation - a set of instructions to produce the desired output document is produced, taking into account the final result set queries and the excluded parts of the document. Generally, this involves walking through the virtual document in document order, executing queries as necessary and emitting elements and attributes.

XML programs can also be recursive, which involves using the same document fragment for both the initial fragment and a set of repeated fragments (each a new query) until some termination criteria or limit is met.

XQuery

XQuery is eligible for specific optimizations. Document projection is the most common optimization. It will be shown in the debug plan as an annotation. For example with the user query containing "xmltable('/a/b' passing doc columns x string path '@x', val string path '/.')", the debug plan would show a tree of the document that will effectively be used by the context and path XQuerys:

```
MEDIUM XQuery Planning Projection conditions met for /a/b - Document projection will be used
childelement(Q{}a)
  childelement(Q{}b)
    attributeattribute(Q{}x)
      childtext()
    childtext()
```

Query Plans

When integrating information using a federated query planner it is useful to view the query plans to better understand how information is being accessed and processed, and to troubleshoot problems.

A query plan (also known as an execution or processing plan) is a set of instructions created by a query engine for executing a command submitted by a user or application. The purpose of the query plan is to execute the user's query in as efficient a way as possible.

Getting a Query Plan

You can get a query plan any time you execute a command. The SQL options available are as follows:

SET SHOWPLAN [ON|DEBUG]- Returns the processing plan or the plan and the full planner Debug Log. See also the SET Statement.

With the above options, the query plan is available from the Statement object by casting to the `org.teiid.jdbc.TeiidStatement` interface or by using the SHOW PLAN statement.

Retrieving a Query Plan Using Teiid Extensions

```
statement.execute("set showplan on");
ResultSet rs = statement.executeQuery("select ...");
TeiidStatement tstatement = statement.unwrap(TeiidStatement.class);
PlanNode queryPlan = tstatement.getPlanDescription();
System.out.println(queryPlan);
```

Retrieving a Query Plan Using Statements

```
statement.execute("set showplan on");
ResultSet rs = statement.executeQuery("select ...");
...
ResultSet planRs = statement.executeQuery("show plan");
planRs.next();
System.out.println(planRs.getString("PLAN_XML"));
```

The query plan is made available automatically in several of Teiid's tools.

Analyzing a Query Plan

Once a query plan has been obtained you will most commonly be looking for:

- Source pushdown – what parts of the query that got pushed to each source
  - Ensure that any predicates especially against indexes are pushed
- Joins - as federated joins can be quite expensive
  - Join ordering - typically influenced by costing
  - Join criteria type mismatches.
  - Join algorithm used - merge, enhanced merge, nested loop, etc.
- Presence of federated optimizations, such as dependent joins.

- Ensure hints have the desired affects - see Hints and Options, hints in the FROM Clause, Subquery Optimization, and Federated Optimizations.

All of the above information can be determined from the processing plan. You will typically be interested in analyzing the textual form of the final processing plan. To understand why particular decisions are made for debugging or support you will want to obtain the full debug log which will contain the intermediate planning steps as well as annotations as to why specific pushdown decisions are made.

A query plan consists of a set of nodes organized in a tree structure. If you are executing a procedure or generating an XML document from an XML Document Model, the overall query plan will contain additional information related the surrounding procedural execution.

In a procedural context the ordering of child nodes implies the order of execution. In most other situation, child nodes may be executed in any order even in parallel. Only in specific optimizations, such as dependent join, will the children of a join execute serially.

Relational Query Plans

Relational plans represent the processing plan that is composed of nodes representing building blocks of logical relational operations. Relational processing plans differ from logical debug relational plans in that they will contain additional operations and execution specifics that were chosen by the optimizer.

The nodes for a relational query plan are:

- Access - Access a source. A source query is sent to the connection factory associated with the source. (For a dependent join, this node is called Dependent Access.)
- Dependent Procedure Access - Access a stored procedure on a source using multiple sets of input values.
- Batched Update - Processes a set of updates as a batch.
- Project - Defines the columns returned from the node. This does not alter the number of records returned.
- Project Into - Like a normal project, but outputs rows into a target table.
- Insert Plan Execution - Similar to a project into, but executes a plan rather than a source query. Typically created when executing an insert into view with a query expression.
- Window Function Project - Like a normal project, but includes window functions.
- Select - Select is a criteria evaluation filter node (WHERE / HAVING).
- Join - Defines the join type, join criteria, and join strategy (merge or nested loop).
- Union All - There are no properties for this node, it just passes rows through from it’s children. Depending upon other factors, such as if there is a transaction or the source query concurrency allowed, not all of the union children will execute in parallel.
- Sort - Defines the columns to sort on, the sort direction for each column, and whether to remove duplicates or not.
- Dup Remove - Removes duplicate rows. The processing uses a tree structure to detect duplicates so that results will effectively stream at the cost of IO operations.
- Grouping - Groups sets of rows into groups and evaluates aggregate functions.
- Null - A node that produces no rows. Usually replaces a Select node where the criteria is always false (and whatever tree is underneath). There are no properties for this node.
- Plan Execution - Executes another sub plan. Typically the sub plan will be a non-relational plan.

- Dependent Procedure Execution - Executes a sub plan using multiple sets of input values.
- Limit - Returns a specified number of rows, then stops processing. Also processes an offset if present.
- XML Table - Evaluates XMLTABLE. The debug plan will contain more information about the XQuery/XPath with regards to their optimization - see the XQuery section below or XQuery Optimization.
- Text Table - Evaluates TEXTTABLE
- Array Table - Evaluates ARRAYTABLE
- Object Table - Evaluates OBJECTTABLE

Node Statistics

Every node has a set of statistics that are output. These can be used to determine the amount of data flowing through the node. Before execution a processor plan will not contain node statistics. Also the statistics are updated as the plan is processed, so typically you'll want the final statistics after all rows have been processed by the client.

Statistic	Description	Units
Node Output Rows	Number of records output from the node	count
Node Next Batch Process Time	Time processing in this node only	millisec
Node Cumulative Next Batch Process Time	Time processing in this node + child nodes	millisec
Node Cumulative Process Time	Elapsed time from beginning of processing to end	millisec
Node Next Batch Calls	Number of times a node was called for processing	count
Node Blocks	Number of times a blocked exception was thrown by this node or a child	count

In addition to node statistics, some nodes display cost estimates computed at the node.

Cost Estimates	Description	Units
Estimated Node Cardinality	Estimated number of records that will be output from the node; -1 if unknown	count

The root node will display additional information.

Top level Statistics	Description	Units
Data Bytes Sent	The size of the serialized data result (row and lob values) sent to the client	bytes

Reading a Processor Plan

The query processor plan can be obtained in a plain text or xml format. The plan text format is typically easier to read, while the xml format is easier to process by tooling. When possible tooling should be used to examine the plans as the tree structures can be deeply nested.

Data flows from the leafs of the tree to the root. Sub plans for procedure execution can be shown inline, and are differentiated by different indentation. Given a user query of `SELECT pm1.g1.e1, pm1.g2.e2, pm1.g3.e3 from pm1.g1 inner join (pm1.g2 left outer join pm1.g3 on pm1.g2.e1=pm1.g3.e1) on pm1.g1.e1=pm1.g3.e1`, the text for a processor plan that does not push down the joins would look like:

```
ProjectNode
  + Output Columns:
    0: e1 (string)
    1: e2 (integer)
    2: e3 (boolean)
  + Cost Estimates:Estimated Node Cardinality: -1.0
  + Child 0:
    JoinNode
      + Output Columns:
        0: e1 (string)
        1: e2 (integer)
        2: e3 (boolean)
      + Cost Estimates:Estimated Node Cardinality: -1.0
      + Child 0:
        JoinNode
          + Output Columns:
            0: e1 (string)
            1: e1 (string)
            2: e3 (boolean)
          + Cost Estimates:Estimated Node Cardinality: -1.0
          + Child 0:
            AccessNode
              + Output Columns:e1 (string)
              + Cost Estimates:Estimated Node Cardinality: -1.0
              + Query:SELECT g_0.e1 AS c_0 FROM pm1.g1 AS g_0 ORDER BY c_0
              + Model Name:pm1
          + Child 1:
            AccessNode
              + Output Columns:
                0: e1 (string)
                1: e3 (boolean)
              + Cost Estimates:Estimated Node Cardinality: -1.0
              + Query:SELECT g_0.e1 AS c_0, g_0.e3 AS c_1 FROM pm1.g3 AS g_0 ORDER BY c_0
              + Model Name:pm1
          + Join Strategy:MERGE JOIN (ALREADY_SORTED/ALREADY_SORTED)
          + Join Type:INNER JOIN
          + Join Criteria:pm1.g1.e1=pm1.g3.e1
      + Child 1:
        AccessNode
          + Output Columns:
            0: e1 (string)
            1: e2 (integer)
          + Cost Estimates:Estimated Node Cardinality: -1.0
          + Query:SELECT g_0.e1 AS c_0, g_0.e2 AS c_1 FROM pm1.g2 AS g_0 ORDER BY c_0
          + Model Name:pm1
      + Join Strategy:ENHANCED SORT JOIN (SORT/ALREADY_SORTED)
      + Join Type:INNER JOIN
      + Join Criteria:pm1.g3.e1=pm1.g2.e1
  + Select Columns:
    0: pm1.g1.e1
    1: pm1.g2.e2
    2: pm1.g3.e3
```

Note that the nested join node is using a merge join and expects the source queries from each side to produce the expected ordering for the join. The parent join is an enhanced sort join which can delay the decision to perform sorting based upon the incoming rows. Note that the outer join from the user query has been modified to an inner join since none of the null inner values can be present in the query result.

The same plan in xml form looks like:

```
<?xml version="1.0" encoding="UTF-8"?>
<node name="ProjectNode">
    <property name="Output Columns">
        <value>e1 (string)</value>
        <value>e2 (integer)</value>
        <value>e3 (boolean)</value>
    </property>
    <property name="Cost Estimates">
        <value>Estimated Node Cardinality: -1.0</value>
    </property>
    <property name="Child 0">
        <node name="JoinNode">
            <property name="Output Columns">
                <value>e1 (string)</value>
                <value>e2 (integer)</value>
                <value>e3 (boolean)</value>
            </property>
            <property name="Cost Estimates">
                <value>Estimated Node Cardinality: -1.0</value>
            </property>
            <property name="Child 0">
                <node name="JoinNode">
                    <property name="Output Columns">
                        <value>e1 (string)</value>
                        <value>e1 (string)</value>
                        <value>e3 (boolean)</value>
                    </property>
                    <property name="Cost Estimates">
                        <value>Estimated Node Cardinality: -1.0</value>
                    </property>
                    <property name="Child 0">
                        <node name="AccessNode">
                            <property name="Output Columns">
                                <value>e1 (string)</value>
                            </property>
                            <property name="Cost Estimates">
                                <value>Estimated Node Cardinality: -1.0</value>
                            </property>
                            <property name="Query">
                                <value>SELECT g_0.e1 AS c_0 FROM pm1.g1 AS g_0 ORDER BY c_0</value>
                            </property>
                            <property name="Model Name">
                                <value>pm1</value>
                            </property>
                        </node>
                    </property>
                    <property name="Child 1">
                        <node name="AccessNode">
                            <property name="Output Columns">
                                <value>e1 (string)</value>
                                <value>e3 (boolean)</value>
                            </property>
                            <property name="Cost Estimates">
                                <value>Estimated Node Cardinality: -1.0</value>
                            </property>
                            <property name="Query">
                                <value>SELECT g_0.e1 AS c_0, g_0.e3 AS c_1 FROM pm1.g3 AS g_0
                                    ORDER BY c_0</value>
                            </property>
                            <property name="Model Name">
```

```
                            <value>pm1</value>
                        </property>
                    </node>
                </property>
                <property name="Join Strategy">
                    <value>MERGE JOIN (ALREADY_SORTED/ALREADY_SORTED)</value>
                </property>
                <property name="Join Type">
                    <value>INNER JOIN</value>
                </property>
                <property name="Join Criteria">
                    <value>pm1.g1.e1=pm1.g3.e1</value>
                </property>
            </node>
        </property>
        <property name="Child 1">
            <node name="AccessNode">
                <property name="Output Columns">
                    <value>e1 (string)</value>
                    <value>e2 (integer)</value>
                </property>
                <property name="Cost Estimates">
                    <value>Estimated Node Cardinality: -1.0</value>
                </property>
                <property name="Query">
                    <value>SELECT g_0.e1 AS c_0, g_0.e2 AS c_1 FROM pm1.g2 AS g_0
                        ORDER BY c_0</value>
                </property>
                <property name="Model Name">
                    <value>pm1</value>
                </property>
            </node>
        </property>
        <property name="Join Strategy">
            <value>ENHANCED SORT JOIN (SORT/ALREADY_SORTED)</value>
        </property>
        <property name="Join Type">
            <value>INNER JOIN</value>
        </property>
        <property name="Join Criteria">
            <value>pm1.g3.e1=pm1.g2.e1</value>
        </property>
    </node>
</property>
<property name="Select Columns">
    <value>pm1.g1.e1</value>
    <value>pm1.g2.e2</value>
    <value>pm1.g3.e3</value>
</property>
</node>
```

Note that the same information appears in each of the plan forms. In some cases it can actually be easier to follow the simplified format of the debug plan final processor plan. From the Debug Log the same plan as above would appear as:

```
OPTIMIZATION COMPLETE:
PROCESSOR PLAN:
ProjectNode(0) output=[pm1.g1.e1, pm1.g2.e2, pm1.g3.e3] [pm1.g1.e1, pm1.g2.e2, pm1.g3.e3]
  JoinNode(1) [ENHANCED SORT JOIN (SORT/ALREADY_SORTED)] [INNER JOIN] criteria=[pm1.g3.e1=pm1.g2.e1] output=[pm
1.g1.e1, pm1.g2.e2, pm1.g3.e3]
    JoinNode(2) [MERGE JOIN (ALREADY_SORTED/ALREADY_SORTED)] [INNER JOIN] criteria=[pm1.g1.e1=pm1.g3.e1] output
=[pm1.g3.e1, pm1.g1.e1, pm1.g3.e3]
      AccessNode(3) output=[pm1.g1.e1] SELECT g_0.e1 AS c_0 FROM pm1.g1 AS g_0 ORDER BY c_0
      AccessNode(4) output=[pm1.g3.e1, pm1.g3.e3] SELECT g_0.e1 AS c_0, g_0.e3 AS c_1 FROM pm1.g3 AS g_0 ORDER
BY c_0
    AccessNode(5) output=[pm1.g2.e1, pm1.g2.e2] SELECT g_0.e1 AS c_0, g_0.e2 AS c_1 FROM pm1.g2 AS g_0 ORDER BY
 c_0
```

Node Properties

Common

- Output Columns - what columns make up the tuples returned by this node
- Data Bytes Sent - how many data byte, not including messaging overhead, were sent by this query
- Planning Time - the amount of time in milliseconds spent planning the query

Relational

- Relational Node ID - matches the node ids seen in the debug log Node(id)
- Criteria - the boolean expression used for filtering
- Select Columns - the columns that define the projection
- Grouping Columns - the columns used for grouping
- Grouping Mapping - shows the mapping of aggregate and grouping column internal names to their expression form
- Query - the source query
- Model Name - the model name
- Sharing ID - nodes sharing the same source results will have the same sharing id
- Dependent Join - if a dependent join is being used
- Join Strategy - the join strategy (Nested Loop, Sort Merge, Enhanced Sort, etc.)
- Join Type - the join type (Left Outer Join, Inner Join, Cross Join)
- Join Criteria - the join predicates
- Execution Plan - the nested execution plan
- Into Target - the insertion target
- Upsert - if the insert is an upsert
- Sort Columns - the columns for sorting
- Sort Mode - if the sort performs another function as well, such as distinct removal
- Rollup - if the group by has the rollup option
- Statistics - the processing statistics
- Cost Estimates - the cost/cardinality estimates including dependent join cost estimates
- Row Offset - the row offset expression
- Row Limit - the row limit expression
- With - the with clause
- Window Functions - the window functions being computed
- Table Function - the table function (XMLTABLE, OBJECTTABLE, TEXTTABLE, etc.)
- Streaming - if the XMLTABLE is using stream processing

XML Plan

- Message
- Tag
- Namespace
- Data Column
- Namespace Declarations
- Optional Flag
- Default Value
- Recursion Direction
- Bindings
- Is Staging Flag
- Source In Memory Flag
- Condition
- Default Program
- Encoding
- Formatted Flag

Procedure

- Expression
- Result Set
- Program
- Variable
- Then
- Else

Other Plans

XML document model queries and procedure execution (including instead of triggers) use intermediate and final plan forms that include relational plans. Generally the structure of the xml/procedure plans will closely match their logical forms. It’s the nested relational plans that will be of interest when analyzing performance issues.

Federated Optimizations

Access Patterns

Access patterns are used on both physical tables and views to specify the need for criteria against a set of columns. Failure to supply the criteria will result in a planning error, rather than a run-away source query. Access patterns can be applied in a set such that only one of the access patterns is required to be satisfied.

Currently any form of criteria referencing an affected column may satisfy an access pattern.

Pushdown

In federated database systems pushdown refers to decomposing the user level query into source queries that perform as much work as possible on their respective source system. Pushdown analysis requires knowledge of source system capabilities, which is provided to Teiid though the Connector API. Any work not performed at the source is then processed in Federate's relational engine.

Based upon capabilities, Teiid will manipulate the query plan to ensure that each source performs as much joining, filtering, grouping, etc. as possible. In may cases, such as with join ordering, planning is a combination of Standard Relational Techniques and, cost based and heuristics for pushdown optimization.

Criteria and join push down are typically the most important aspects of the query to push down when performance is a concern. See Query Plans on how to read a plan to ensure that source queries are as efficient as possible.

Dependent Joins

A special optimization called a dependent join is used to reduce the rows returned from one of the two relations involved in a multi-source join. In a dependent join, queries are issued to each source sequentially rather than in parallel, with the results obtained from the first source used to restrict the records returned from the second. Dependent joins can perform some joins much faster by drastically reducing the amount of data retrieved from the second source and the number of join comparisons that must be performed.

The conditions when a dependent join is used are determined by the query planner based on Access Patterns, hints, and costing information. There are three different kinds of dependent joins that Teiid supports:

- Join based on in/equality support - where the engine will determine how to break up large queries based upon translator capabilities
- Key Pushdown - where the translator has access to the full set of key values and determines what queries to send
- Full Pushdown - where translator ships the all data from the independent side to the translator. Can be used automatically by costing or can be specified as an option in the hint.

Teiid supports hints to control dependent join behavior:

- MAKEIND - indicates that the clause should be the independent side of a dependent join.
- MAKEDEP - indicates that the clause should be the dependent side of a join. MAKEDEP as a non-comment hint supports optional max and join arguments - MAKEDEP(JOIN) meaning that the entire join should be pushed, and MAKEDEP(MAX:5000) meaning that the dependent join should only be performed if there are less than the max number of values from the independent side.

- MAKENOTDEP - prevents the clause from being the dependent side of a join.

Theses can be placed in either the OPTION Clause or directly in the FROM Clause. As long as all Access Patterns can be met, the MAKEIND, MAKEDEP, and MAKENOTDEP hints override any use of costing information. MAKENOTDEP supersedes the other hints.

Tip	The MAKEDEP/MAKEIND hint should only be used if the proper query plan is not chosen by default. You should ensure that your costing information is representative of the actual source cardinality.

Note	An inappropriate MAKEDEP/MAKEIND hint can force an inefficient join structure and may result in many source queries.

Tip	While these hints can be applied to views, the optimizer will by default remove views when possible. This can result in the hint placement being significantly different than the original intention. You should consider using the NO_UNNEST hint to prevent the optimizer from removing the view in these cases.

In the simplest scenario the engine will use IN clauses (or just equality predicates) to filter the values coming from the dependent side. If the number of values from the independent side exceeds the translators MaxInCriteriaSize, the values will be split into multiple IN predicates up to MaxDependentPredicates. When the number of independent values exceeds MaxInCriteriaSize*MaxDependentPredicates, then multiple dependent queries will be issued in parallel.

If the translator returns true for supportsDependentJoins, then the engine may provide the entire set of independent key values. This will occur when the number of independent values exceeds MaxInCriteriaSize*MaxDependentPredicates so that the translator may use specific logic to avoid issuing multiple queries as would happen in the simple scenario.

If the translator returns true for both supportsDependentJoins and supportsFullDependentJoins then a full pushdown may be chosen by the optimizer A full pushdown, sometimes also called as data-ship pushdown, is where all the data from independent side of the join is sent to dependent side. This has an added benefit of allowing the plan above the join to be eligible for pushdown as well. This is why the optimizer may choose to perform a full pushdown even when the number of independent values does not exceed MaxInCriteriaSize*MaxDependentPredicates. You may also force full pushdown using the MAKEDEP(JOIN) hint. The translator is typically responsible for creating, populating, and removing a temporary table that represents the independent side. If you are working with custom translators see Dependent Join Pushdown as to how to support it key and full pushdown.

Note	**Key/Full Pushdown** is currently only supported out-of-the box by a subset of JDBC translators. To enable support, set the translator override property "enableDependentJoins" to "true". The JDBC source must support the creation of temporary tables, which typically requires a Hibernate dialect. Translators that should support this feature include: DB2, Derby, H2, Hana, HSQL, MySQL, Oracle, PostgreSQL, SQL Server, 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

The optional join hint indicates to the optimizer that a joined table should be omitted if none of its columns are used by the output of the user query or in a meaningful way to construct the results of the user query. This hint is typically only used in view layers containing multi-source joins.

The optional join hint is applied as a comment on a join clause. It can be applied in both ANSI and non-ANSI joins. With non-ANSI joins an entire joined table may be marked as optional.

Example Optional Join Hint

```
select a.column1, b.column2 from a, /*+ optional */ b WHERE a.key = b.key
```

Suppose this example defines a view layer X. If X is queried in such a way as to not need b.column2, then the optional join hint will cause b to be omitted from the query plan. The result would be the same as if X were defined as:

Example Optional Join Hint

```
select a.column1 from a
```

Example ANSI Optional Join Hint

```
select a.column1, b.column2, c.column3 from /*+ optional */ (a inner join b ON a.key = b.key) INNER JOIN c ON a.
key = c.key
```

In this example the ANSI join syntax allows for the join of a and b to be marked as optional. Suppose this example defines a view layer X. Only if both column a.column1 and b.column2 are not needed, e.g. "SELECT column3 FROM X" will the join be removed.

The optional join hint will not remove a bridging table that is still required.

Example Bridging Table

```
select a.column1, b.column2, c.column3 from /*+ optional */ a, b, c WHERE ON a.key = b.key AND a.key = c.key
```

Suppose this example defines a view layer X. If b.column2 or c.column3 are solely required by a query to X, then the join on a be removed. However if a.column1 or both b.column2 and c.column3 are needed, then the optional join hint will not take effect.

When a join clause is omitted via the optional join hint, the relevant criteria is not applied. Thus it is possible that the query results may not have the same cardinality or even the same row values as when the join is fully applied.

Left/right outer joins where the inner side values are not used and whose rows under go a distinct operation will automatically be treated as an optional join and do not require a hint.

Example Unnecessary Optional Join Hint

```
    select distinct a.column1 from a LEFT OUTER JOIN /*+optional*/ b ON a.key = b.key
```

Note	A simple "SELECT COUNT(*) FROM VIEW" against a view where all join tables are marked as optional will not return a meaningful result.

Source Hints

Teiid user and transformation queries can contain a meta source hint that can provide additional information to source queries. The source hint has the form:

```
/*+ sh[[ KEEP ALIASES]:'arg'] source-name[ KEEP ALIASES]:'arg1' ... */
```

- The source hint is expected to appear after the query (SELECT, INSERT, UPDATE, DELETE) keyword.
- Source hints may appear in any subquery or in views. All hints applicable to a given source query will be collected and pushed down together as a list. The order of the hints is not guaranteed.
- The sh arg is optional and is passed to all source queries via the `ExecutionContext.getGeneralHints` method. The additional args should have a source-name that matches the source name assigned to the translator in the VDB. If the source-name matches, the hint values will be supplied via the `ExecutionContext.getSourceHints` method. See the Developer's Guide for more on using an ExecutionContext.
- Each of the arg values has the form of a string literal - it must be surrounded in single quotes and a single quote can be escaped with another single quote. Only the Oracle translator does anything with source hints by default. The Oracle translator will use both the source hint and the general hint (in that order) if available to form an Oracle hint enclosed in /*+ … */.
- If the KEEP ALIASES option is used either for the general hint or on the applicable source specific hint, then the table/view aliases from the user query and any nested views will be preserved in the push-down query. This is useful in situations where the source hint may need to reference aliases and the user does not wish to rely on the generated aliases (which can be seen in the query plan in the relevant source queries - see above). However in some situations this may result in an invalid source query if the preserved alias names are not valid for the source or result in a name collision. If the usage of KEEP ALIASES results in an error, the query could be modified by preventing view removal with the NO_UNNEST hint, the aliases modified, or the KEEP ALIASES option could be removed and the query plan used to determine the generated alias names.

Sample Source Hints

```
SELECT /*+ sh:'general hint' */ ...

SELECT /*+ sh KEEP ALIASES:'general hint' my-oracle:'oracle hint' */ ...
```

Partitioned Union

Union partitioning is inferred from the transformation/inline view. If one (or more) of the UNION columns is defined by constants and/or has WHERE clause IN predicates containing only constants that make each branch mutually exclusive, then the UNION is considered partitioned. UNION ALL must be used and the UNION cannot have a LIMIT, WITH, or ORDER BY clause (although individual branches may use LIMIT, WITH, or ORDER BY). Partitioning values should not be null. For example the view definition "select 1 as x, y from foo union all select z, a from foo1 where z in (2, 3)" would be considered 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 that more advanced or explicit partition could be considered in the future. The concept of a partitioned union is used for performing partition-wise joins, in Updatable Views, and Partial Aggregate Pushdown.

Standard Relational Techniques

Teiid also incorporates many standard relational techniques to ensure efficient query plans.

- Rewrite analysis for function simplification and evaluation.
- Boolean optimizations for basic criteria simplification.
- Removal of unnecessary view layers.

- Removal of unnecessary sort operations.
- Advanced search techniques through the left-linear space of join trees.
- Parallelizing of source access during execution.
- Subquery Optimization

Join Compensation

Some source systems only allow "relationship" queries logically producing left outer join results even when queried with an inner join Teiid will attempt to form an appropriate left outer join. These sources are restricted to only supporting key joins. In some circumstances foreign key relationships on the same source should not be traversed at all or with the referenced table on the outer side of join. The extension property teiid_rel:allow-join can be used on the foreign key to further restrict the pushdown behavior. With a value of "false" no join pushdown will be allowed, and with a value of "inner" the referenced table must be on the inner side of the join. If the join pushdown is prevented, the join will be processed as a federated join.

Subquery Optimization

- EXISTS subqueries are typically rewrite to "SELECT 1 FROM …" to prevent unnecessary evaluation of SELECT expressions.
- Quantified compare SOME subqueries are always turned into an equivalent IN predicate or comparison against an aggregate value. e.g. col > SOME (select col1 from table) would become col > (select min(col1) from table)
- Uncorrelated EXISTs and scalar subquery that are not pushed to the source can be pre-evaluated prior to source command formation.
- Correlated subqueries used in DETELEs or UPDATEs that are not pushed as part of the corresponding DELETE/UPDATE will cause Teiid to perform row-by-row compensating processing. This will only happen if the affected table has a primary key. If it does not, then an exception will be thrown.
- WHERE or HAVING clause IN, Quantified Comparison, Scalar Subquery Compare, and EXISTs predicates can take the MJ (merge join), DJ (dependent join), or NO_UNNEST (no unnest) hints appearing just before the subquery. The MJ hint directs the optimizer to use a traditional, semijoin, or antisemijoin merge join if possible. The DJ is the same as the MJ hint, but additionally directs the optimizer to use the subquery as the independent side of a dependent join if possible. The NO_UNNEST hint, which supersedes the other hints, will direct the optimizer to leave the subquery in place.
- SELECT scalar subqueries can take the MJ (merge join) or NO_UNNEST (no unnest) hints appearing just before the subquery. The MJ hint directs the optimizer to use a traditional or semijoin merge join if possible. The NO_UNNEST hint, which supercedes 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.
- Conversion of subquery predicates to nested loop joins is not yet available.

XQuery Optimization

A technique known as document projection is used to reduce the memory footprint of the context item document. Document projection loads only the parts of the document needed by the relevant XQuery and path expressions. Since document projection analysis uses all relevant path expressions, even 1 expression that could potentially use many nodes, e.g. //x rather than /a/b/x will cause a larger memory footprint. With the relevant content removed the entire document will still be loaded into memory for processing. Document projection will only be used when there is a context item (unnamed PASSING clause item) passed to XMLTABLE/XMLQUERY. A named variable will not have document projection performed. In some cases the expressions used may be too complex for the optimizer to use document projection. You should check the SHOWPLAN DEBUG full plan output to see if the appropriate optimization has been performed.

With additional restrictions, simple context path expressions allow the processor to evaluate document subtrees independently - without loading the full document in memory. A simple context path expression can be of the form "[/][ns:]root/[ns1:]elem/…", where a namespace prefix or element name can also be the * wild card. As with normal XQuery processing if namespace prefixes are used in the XQuery expression, they should be declared using the XMLNAMESPACES clause.

Streaming Eligible XMLQUERY

```
XMLQUERY('/*:root/*:child' PASSING doc)
```

Rather than loading the entire doc in-memory as a DOM tree, each child element will be independently added to the result.

Streaming Ineligible XMLQUERY

```
XMLQUERY('//child' PASSING doc)
```

The use of the descendant axis prevents the streaming optimization, but document projection can still be performed.

When using XMLTABLE, the COLUMN PATH expressions have additional restrictions. They are allowed to reference any part of the element subtree formed by the context expression and they may use any attribute value from their direct parentage. Any path expression where it is possible to reference a non-direct ancestor or sibling of the current context item prevent streaming from being used.

Streaming Eligible XMLTABLE

```
XMLTABLE('/*:root/*:child' PASSING doc COLUMNS fullchild XML PATH '.', parent_attr string PATH '../@attr', chil
d_val integer)
```

The context XQuery and the column path expression allow the streaming optimization, rather than loading the entire doc in-memory as a DOM tree, each child element will be independently added to the result.

Streaming Ineligible XMLTABLE

```
XMLTABLE('/*:root/*:child' PASSING doc COLUMNS sibling_attr string PATH '../other_child/@attr')
```

The reference of an element outside of the child subtree in the sibling_attr path prevents the streaming optimization from being used, but document projection can still be performed.

Note	Column paths should be as targeted as possible to avoid performance issues. A general path such as `..//child' will cause the entire subtree of the context item to be searched on each output row.

Partial Results

Teiid provides the capability to obtain "partial results" in the event of data source unavailability or failure. This is especially useful when unioning information from multiple sources, or when doing a left outer join, where you are `appending' columns to a master record but still want the record if the extra information is not available.

A source is considered to be `unavailable' if the connection factory associated with the source issues an exception in response to a query. The exception will be propagated to the query processor, where it will become a warning on the statement. See the Client Guide for more on Partial Results Mode and SQLWarnings.

Conformed Tables

A conformed table is a source table that is the same in several physical sources. Unlike Multisource Models which assume a partitioning paradigm, the planner assumes any conformed table may be substituted for another to improve performance. Typically this would be used when reference data exists in multiple sources, but only a single metadata entry is desired to represent the table.

Conformed tables are defined by adding the

```
{http://www.teiid.org/ext/relational/2012}conformed-sources
```

extension metadata property to the appropriate source tables. Extension properties can be set at design time in Designer, in the vdb.xml when using full DDL Metadata or alter statements, or at runtime using the setProperty system procedure. The property is expected to be a comma separated list of physical model/schema names.

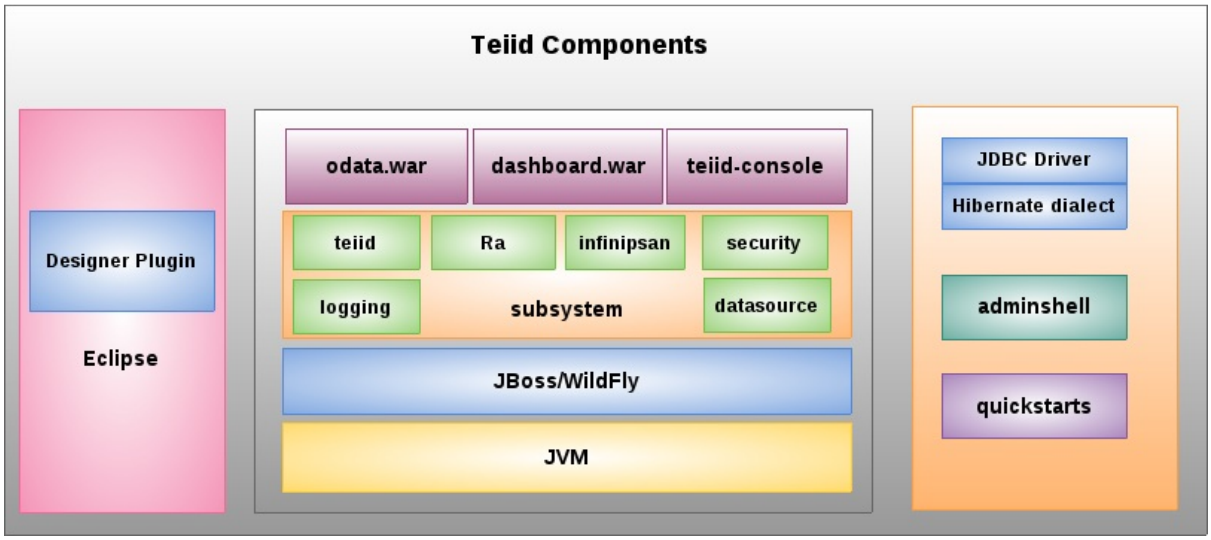
DDL Alter Example

```
ALTER FOREIGN TABLE "reference_data" OPTIONS (ADD "teiid_rel:conformed-sources" 'source2,source3');
```

There is no expectation that a metadata entry exists on the other schemas. Just as with the multi-source feature, there is then no source specific metadata entry to the conformed sources. Also just as with multi-source planning, the capabilities are assumed to be the same across conformed sources.

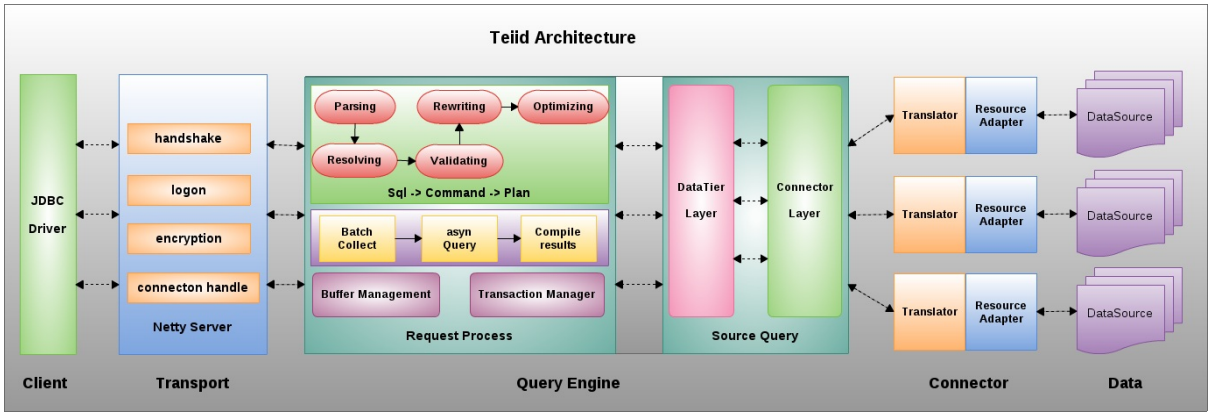
The engine will take the list of conformed sources and associate a set of model metadata ids to the corresponding access node. The logic considering joins and subqueries will also consider the conformed sets when making pushdown decisions. The subquery handling will only check for conformed sources for the subquery - not in the parent. So having a conformed table in the subquery will pushdown as expected, but not vice versa.

Teiid Components



- **Designer Plugin** - Eclipse Plugin based Teiid design environment, used to connect/federate/transform datasources to produce a `.vdb` file.
- **JVM** - Teiid is a pure Java Data Virtualization Platform.
- **WildFly** - Teiid use a plugable installation which need a WildFly Server installed, alternatively, a full installed WildFly kit be distributed.
- **Subsystem** - Due to WildFly's Modular and Pluggable Architecture(a series of Management commands compose of a subsystem, a series of subsystems compose of the whole server), Teiid implement WildFly's Controller/Management API developed a `teiid` subsystem and reuse lots of other subsystems like `resource-adapter`, `infinispan`, `security`, `logging`, `datasource`.
- **odata.war** - Teiid support OData via odata.war, more details refer to OData Support
- **dashboard.war** - A web based dashboard generator.
- **teiid-console** - A web based administrative and monitoring tool for Teiid, more details refer to Teiid Console
- **JDBC Driver** - JDBC Driver to connect to Teiid Server.
- **adminshell** - A scripting based Monitor/Management Tool, more details refer to AdminShell
- **quickstarts** - A maven quickstart showing how to utilize Teiid.

Teiid Architecture



- **Client** - Client Develop Guide
- **Transport** - A Transport managements client connections - security authentication, encryption, etc.
- **Query Engine** - The Query Engine has several layers / components. Request processing at a high level:

    1. SQL is converted to a Processor Plan. The engine receives an incoming SQL query. It is parsed to a internal command. Then the command is converted a logical plan via resolving, validating, and rewriting. Lastly rule and cost-based optimization convert the logical plan to a final Processor Plan. More details refer to Federated Planning.
    2. Batch Processing. The source and other aspects of query processing may return results asynchronously to the processing thread. As soon as possible batches of results are made available to the client.
    3. Buffer Management Controls the bulk of the on and off heap memory that Teiid is using. It prevents consuming too much memory that otherwise might exceed the vm size.
    4. Transaction Management determines when transactions are needed and interacts with the TransactionManager subsystem to coordinate XA transactions.

Source queries are handled by the Data Tier layer which interfaces with the Query Engine and the Connector Layer which utilizes a Translator/Resource Adapter pair to interact directly with a source. Connectivity is provided for heterogeneous data stores, like Databases/Data warehouse, NoSQL, Hadoop, Data Grid/Cache, File, SaaS, etc. - see Data Sources.

- **Translator** - Teiid has developed a series of Translators, for more details refer to Translators.
- **Resource Adapter** - Provides container managed access to a source, for more details refer to Developing JEE Connectors.

Terminology

- VM or Process – a JBossAS instance running Teiid.
- Host – a machine that is "hosting" one or more VMs.
- Service – a subsystem running in a VM (often in many VMs) and providing a related set of functionality. In addition to these main components, the service platform provides a core set of services available to applications built on top of the service platform. These services are:
- Session – the Session service manages active session information.
- Buffer Manager – the Buffer Manager service provides access to data management for intermediate results.
- Transaction – the Transaction service manages global, local, and request scoped transactions. See also the documentation on Transaction Support.

Data Management

Cursoring and Batching

Teiid cursors all results, regardless of whether they are from one source or many sources, and regardless of what type of processing (joins, unions, etc.) have been performed on the results.

Teiid processes results in batches. A batch is simply a set of records. The number of rows in a batch is determined by the buffer system property *processor-batch-size* and is scaled upon the estimated memory footprint of the batch.

Client applications have no direct knowledge of batches or batch sizes, but rather specify fetch size. However the first batch, regardless of fetch size is always proactively returned to synchronous clients. Subsequent batches are returned based on client demand for the data. Pre-fetching is utilized at both the client and connector levels.

Buffer Management

The buffer manager manages memory for all result sets used in the query engine. That includes result sets read from a connection factory, result sets used temporarily during processing, and result sets prepared for a user. Each result set is referred to in the buffer manager as a tuple source.

When retrieving batches from the buffer manager, the size of a batch in bytes is estimated and then allocated against the max limit.

Memory Management

The buffer manager has two storage managers - a memory manager and a disk manager. The buffer manager maintains the state of all the batches, and determines when batches must be moved from memory to disk.

Disk Management

Each tuple source has a dedicated file (named by the ID) on disk. This file will be created only if at least one batch for the tuple source had to be swapped to disk. The file is random access. The processor batch size property defines how many rows should nominally exist in a batch assuming 2048 bits worth of data in a row. If the row is larger or smaller than that target, the engine will adjust the batch size for those tuples accordingly. Batches are always read and written from the storage manager whole.

The disk storage manager has a cap on the maximum number of open files to prevent running out of file handles. In cases with heavy buffering, this can cause wait times while waiting for a file handle to become available (the default max open files is 64).

Cleanup

When a tuple source is no longer needed, it is removed from the buffer manager. The buffer manager will remove it from both the memory storage manager and the disk storage manager. The disk storage manager will delete the file. In addition, every tuple source is tagged with a "group name" which is typically the session ID of the client. When the client’s session is terminated (by closing the connection, server detecting client shutdown, or administrative termination), a call is sent to the buffer manager to remove all tuple sources for the session.

In addition, when the query engine is shutdown, the buffer manager is shut down, which will remove all state from the disk storage manager and cause all files to be closed. When the query engine is stopped, it is safe to delete any files in the buffer directory as they are not used across query engine restarts and must be due to a system crash where buffer files were not cleaned

up.

Query Termination

Canceling Queries

When a query is canceled, processing will be stopped in the query engine and in all connectors involved in the query. The semantics of what a connector does in response to a cancellation command is dependent on the connector implementation. For example, JDBC connectors will asynchronously call cancel on the underlying JDBC driver, which may or may not actually support this method.

User Query Timeouts

User query timeouts in Teiid can be managed on the client-side or the server-side. Timeouts are only relevant for the first record returned. If the first record has not been received by the client within the specified timeout period, a `cancel' command is issued to the server for the request and no results are returned to the client. The cancel command is issued asynchronously without the client’s intervention.

The JDBC API uses the query timeout set by the `java.sql.Statement.setQueryTimeout` method. You may also set a default statement timeout via the connection property "QUERYTIMEOUT". ODBC clients may also utilize QUERYTIMEOUT as an execution property via a set statement to control the default timeout setting. See the Client Developers Guide for more on connection/execution properties and set statements.

Server-side timeouts start when the query is received by the engine. There may be a skew from the when the client issued the query due to network latency or server load that may slow the processing of IO work. The timeout will be cancelled if the first result is sent back before the timeout has ended. See the VDBs section for more on setting the query-timeout VDB property. See the Admin Guide for more on modifying the file to set default query timeout for all queries.

Processing

Join Algorithms

Nested loop does the most obvious processing – for every row in the outer source, it compares with every row in the inner source. Nested loop is only used when the join criteria has no equi-join predicates.

Merge join first sorts the input sources on the joined columns. You can then walk through each side in parallel (effectively one pass through each sorted source) and when you have a match, emit a row. In general, merge join is on the order of n+m rather than n*m in nested loop. Merge join is the default algorithm.

Using costing information the engine may also delay the decision to perform a full sort merge join. Based upon the actual row counts involved, the engine can choose to build an index of the smaller side (which will perform similarly to a hash join) or to only partially sort the larger side of the relation.

Joins involving equi-join predicates are also eligible to be made into dependent joins.

Sort Based Algorithms

Sorting is used as the basis of the Sort (ORDER BY), Grouping (GROUP BY), and DupRemoval (SELECT DISTINCT) operations. The sort algorithm is a multi-pass merge-sort that does not require all of the result set to ever be in memory yet uses the maximal amount of memory allowed by the buffer manager.

It consists of two phases. The first phase ("sort") will take an unsorted input stream and produce one or more sorted input streams. Each pass reads as much of the unsorted stream as possible, sorts it, and writes it back out as a new stream. Since the stream may be more than can fit in memory, this may result in many sorted streams.

The second phase ("merge") consists of a set of phases that grab the next batch from as many sorted input streams as will fit in memory. It then repeatedly grabs the next tuple in sorted order from each stream and outputs merged sorted batches to a new sorted stream. At completion of the pass, all input streams are dropped. In this way, each pass reduces the number of sorted streams. When only one stream remains, it is the final output.

BNF for SQL Grammar

- Main Entry Points
  - callable statement
  - ddl statement
  - procedure body definition
  - directly executable statement
- Reserved Keywords
- Non-Reserved Keywords
- Reserved Keywords For Future Use
- Tokens
- Production Cross-Reference
- Productions

Reserved Keywords

Keyword	Usage
ADD	add set child option, add set option, ADD column
ALL	standard aggregate function, function, query expression body, query term, grant type, select clause, quantified comparison predicate
ALTER	alter, alter column options, alterStatement, grant type
AND	between predicate, boolean term
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, option namespace, create table, create trigger, derived column, dynamic data statement, function, loop statement, xml namespace element, object table, select derived column, table subquery, text table, table name, unescapedFunction, with list element, xml serialize, xml table
ASC	sort specification
ATOMIC	compound statement, for each row trigger action

AUTHENTICATED	with role
BEGIN	compound statement, for each row trigger action
BETWEEN	between predicate
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, alter column options, DROP column, rename column options, Create GRANT, Revoke GRANT
COMMIT	create temporary table
CONSTRAINT	create table body, Create GRANT, Revoke GRANT
CONTINUE	branching statement
CONVERT	function
CREATE	create procedure, create data wrapper aka translator, create database, create foreign temp table, create role, create schema, create server, aka data source, create table, create temporary table, create trigger, procedure body definition
CROSS	cross join
DATE	non numeric literal, simple data type

DAY	function
DECIMAL	simple data type
DECLARE	declare statement
DEFAULT	table element, xml namespace element, object table column, procedure parameter, xml table column
DELETE	alter, ALTER TRIGGER, create trigger, delete statement, grant type
DESC	sort specification
DISTINCT	standard aggregate function, function, if statement, query expression body, query term, select clause
DOUBLE	simple data type
DROP	DROP column, drop option, Drop data wrapper aka translator, drop database, drop option, drop procedure, drop role, drop schema, drop server, aka data source, drop table, drop table, grant type
EACH	for each row trigger action
ELSE	case expression, if statement, searched case expression
END	case expression, compound statement, for each row trigger action, searched case expression
ERROR	raise error statement
ESCAPE	match predicate, text table
EXCEPT	Import foreign schema, query expression body
EXEC	dynamic data statement, call statement
EXECUTE	dynamic data statement, grant type, call statement
EXISTS	exists predicate
FALSE	non numeric literal
FETCH	fetch clause
FILTER	filter clause
FLOAT	simple data type
FOR	for each row trigger action, function, text aggreate function, text table column, xml table column

FOREIGN	alterStatement, create procedure, create data wrapper aka translator, create foreign temp table, create server, aka data source, create table, Drop data wrapper aka translator, drop procedure, drop table, foreign key, Import foreign schema
FROM	delete statement, from clause, function, if statement, Import foreign schema, Revoke GRANT
FULL	qualified table
FUNCTION	create procedure, drop procedure
GEOMETRY	simple data type
GLOBAL	create table, drop table
GRANT	Create GRANT, Revoke GRANT
GROUP	group by clause
HAVING	having clause
HOUR	function
IF	if statement
IMMEDIATE	dynamic data statement
IMPORT	Import another Database, Import foreign schema
IN	procedure parameter, in predicate
INNER	qualified table
INOUT	procedure parameter
INSERT	alter, ALTER TRIGGER, create trigger, function, insert statement, grant type
INTEGER	simple data type
INTERSECT	query term
INTO	dynamic data statement, Import foreign schema, insert statement, into clause
IS	if statement, is null predicate
JOIN	cross join, make dep options, qualified table
LANGUAGE	object table, grant type
LATERAL	table subquery

LEADING	function
LEAVE	branching statement
LEFT	function, qualified table
LIKE	match predicate
LIKE_REGEX	like regex predicate
LIMIT	Import foreign schema, 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, if statement, 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, is null predicate, non numeric literal, procedure parameter, procedure result column, temporary table element, xml query
OBJECT	simple data type
OF	alter, ALTER TRIGGER, create trigger
OFFSET	limit clause
ON	alter, ALTER TRIGGER, create foreign temp table, create temporary table, create trigger, Create GRANT, loop statement, qualified table, Revoke GRANT, xml query

ONLY	fetch clause
OPTION	option clause
OPTIONS	alter child options list, alter options list, options clause
OR	boolean value expression
ORDER	Create GRANT, order by clause, Revoke GRANT
OUT	procedure parameter
OUTER	qualified table
OVER	window specification
PARAMETER	alter column options, rename column options
PARTITION	window specification
PRIMARY	table element, create temporary table, primary key
PROCEDURE	alter, ALTER PROCEDURE, create procedure, drop procedure, Create GRANT, procedure body definition, Revoke GRANT
REAL	simple data type
REFERENCES	foreign key
RETURN	assignment statement, return statement, data statement
RETURNS	create procedure
REVOKE	Revoke GRANT
RIGHT	function, qualified table
ROLLUP	group by clause
ROW	fetch clause, for each row trigger action, limit clause, text table
ROWS	create temporary table, fetch clause, limit clause
SECOND	function
SELECT	grant type, select clause
SERVER	ALTER SERVER, create schema, create server, aka data source, drop server, aka data source, Import foreign schema

SET	add set child option, add set option, option namespace, update statement, use schema
SHORT	simple data type
SIMILAR	match predicate
SMALLINT	simple data type
SOME	standard aggregate function, quantified comparison predicate
SQLEXCEPTION	sql exception
SQLSTATE	sql exception
SQLWARNING	raise statement
STRING	dynamic data statement, simple data type, xml serialize
TABLE	ALTER TABLE, create procedure, create foreign temp table, create table, create temporary table, drop table, drop table, Create GRANT, query primary, grant type, Revoke GRANT, table subquery
TEMPORARY	create foreign temp table, create table, create temporary table, drop table, grant type
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, Create GRANT, Import foreign schema, match predicate
TRAILING	function
TRANSLATE	function
TRIGGER	alter, ALTER TRIGGER, create trigger
TRUE	non numeric literal
UNION	cross join, query expression body
UNIQUE	other constraints, table element
UNKNOWN	non numeric literal

UPDATE	alter, ALTER TRIGGER, create trigger, dynamic data statement, grant type, update statement
USER	function
USING	dynamic data statement
VALUES	query primary
VARBINARY	simple data type, xml serialize
VARCHAR	simple data type, xml serialize
VIRTUAL	alterStatement, create procedure, create schema, create table, drop procedure, drop schema, drop table, procedure body definition
WHEN	case expression, searched case expression
WHERE	filter clause, where clause
WHILE	while statement
WITH	assignment statement, create role, Import another Database, query expression, with role, data statement
WITHOUT	assignment statement, data statement
WRAPPER	ALTER DATA WRAPPER, create data wrapper aka translator, create server, aka data source, Drop data wrapper aka translator
XML	simple data type
XMLAGG	ordered aggregate function
XMLATTRIBUTES	xml attributes
XMLCAST	unescapedFunction
XMLCOMMENT	function
XMLCONCAT	function
XMLELEMENT	xml element
XMLEXISTS	xml query
XMLFOREST	xml forest
XMLNAMESPACES	xml namespaces
XMLPARSE	xml parse

XMLPI	function
XMLQUERY	xml query
XMLSERIALIZE	xml serialize
XMLTABLE	xml table
XMLTEXT	function
YEAR	function

Non-Reserved Keywords

Name	Usage
ACCESS	Import another Database, non-reserved identifier
ACCESSPATTERN	other constraints, non-reserved identifier
AFTER	alter, create trigger, non-reserved identifier
ARRAYTABLE	array table, non-reserved identifier
AUTO_INCREMENT	alter column options, table element, non-reserved identifier
AVG	standard aggregate function, non-reserved identifier
CHAIN	sql exception, non-reserved identifier
COLUMNS	array table, non-reserved identifier, object table, text table, xml table
CONDITION	Create GRANT, non-reserved identifier, Revoke GRANT
CONTENT	non-reserved identifier, xml parse, xml serialize
CONTROL	Import another Database, non-reserved identifier
COUNT	standard aggregate function, non-reserved identifier
DATA	ALTER DATA WRAPPER, create data wrapper aka translator, create server, aka data source, Drop data wrapper aka translator, non-reserved identifier
DATABASE	ALTER DATABASE, create database, drop database, Create GRANT, Import another Database, non-reserved identifier, Revoke GRANT, use database
DELIMITER	non-reserved identifier, text aggreate function, text table
DENSE_RANK	analytic aggregate function, non-reserved identifier

DISABLED	alter, ALTER TRIGGER, non-reserved identifier
DOCUMENT	non-reserved identifier, xml parse, xml serialize
EMPTY	non-reserved identifier, xml query
ENABLED	alter, ALTER TRIGGER, non-reserved identifier
ENCODING	non-reserved identifier, text aggreate function, xml serialize
EVERY	standard aggregate function, non-reserved identifier
EXCEPTION	compound statement, declare statement, non-reserved identifier
EXCLUDING	non-reserved identifier, xml serialize
EXTRACT	function, non-reserved identifier
FIRST	fetch clause, non-reserved identifier, sort specification
HEADER	non-reserved identifier, text aggreate function, text table column, text table
INCLUDING	non-reserved identifier, xml serialize
INDEX	other constraints, table element, non-reserved identifier
INSTEAD	alter, ALTER TRIGGER, create trigger, non-reserved identifier
JAAS	non-reserved identifier, with role
JSONARRAY_AGG	non-reserved identifier, ordered aggregate function
JSONOBJECT	json object, non-reserved identifier
KEY	table element, create temporary table, foreign key, non-reserved identifier, primary key
LAST	non-reserved identifier, sort specification
MASK	Create GRANT, non-reserved identifier, Revoke GRANT
MAX	standard aggregate function, make dep options, non-reserved identifier
MIN	standard aggregate function, non-reserved identifier
NAME	function, non-reserved identifier, xml element
NAMESPACE	option namespace, non-reserved identifier

NEXT	fetch clause, non-reserved identifier
NONE	non-reserved identifier
NULLS	non-reserved identifier, sort specification
OBJECTTABLE	non-reserved identifier, object table
ORDINALITY	non-reserved identifier, text table column, xml table column
PASSING	non-reserved identifier, object table, xml query, xml query, xml table
PATH	non-reserved identifier, xml table column
PRESERVE	create temporary table, non-reserved identifier
PRIVILEGES	non-reserved identifier, grant type
QUERYSTRING	non-reserved identifier, querystring function
QUOTE	non-reserved identifier, text aggreate function, text table
RAISE	non-reserved identifier, raise statement
RANK	analytic aggregate function, non-reserved identifier
RENAME	rename column options, non-reserved identifier
REPOSITORY	Import foreign schema, non-reserved identifier
RESULT	non-reserved identifier, procedure parameter
ROLE	create role, drop role, non-reserved identifier, with role
ROW_NUMBER	analytic aggregate function, non-reserved identifier
SCHEMA	create schema, drop schema, Create GRANT, Import foreign schema, non-reserved identifier, Revoke GRANT, use schema
SELECTOR	non-reserved identifier, text table column, text table
SERIAL	alter column options, table element, non-reserved identifier, temporary table element
SKIP	non-reserved identifier, text table
SQL_TSI_DAY	time interval, non-reserved identifier
SQL_TSI_FRAC_SECOND	time interval, non-reserved identifier

SQL_TSI_HOUR	time interval, non-reserved identifier
SQL_TSI_MINUTE	time interval, non-reserved identifier
SQL_TSI_MONTH	time interval, non-reserved identifier
SQL_TSI_QUARTER	time interval, non-reserved identifier
SQL_TSI_SECOND	time interval, non-reserved identifier
SQL_TSI_WEEK	time interval, non-reserved identifier
SQL_TSI_YEAR	time interval, non-reserved identifier
STDDEV_POP	standard aggregate function, non-reserved identifier
STDDEV_SAMP	standard aggregate function, non-reserved identifier
SUBSTRING	function, non-reserved identifier
SUM	standard aggregate function, non-reserved identifier
TEXTAGG	non-reserved identifier, text aggreate function
TEXTTABLE	non-reserved identifier, text table
TIMESTAMPADD	function, non-reserved identifier
TIMESTAMPDIFF	function, non-reserved identifier
TO_BYTES	function, non-reserved identifier
TO_CHARS	function, non-reserved identifier
TRANSLATOR	ALTER DATA WRAPPER, create data wrapper aka translator, create server, aka data source, Drop data wrapper aka translator, non-reserved identifier
TRIM	function, non-reserved identifier, text table column, text table
TYPE	alter column options, create data wrapper aka translator, create server, aka data source, non-reserved identifier
UPSERT	insert statement, non-reserved identifier
USE	non-reserved identifier, use database
VARIADIC	non-reserved identifier, procedure parameter
VAR_POP	standard aggregate function, non-reserved identifier
VAR_SAMP	standard aggregate function, non-reserved identifier

VERSION	create database, create server, aka data source, drop database, Import another Database, non-reserved identifier, use database, xml serialize
VIEW	alter, ALTER TABLE, create table, drop table, non-reserved identifier
WELLFORMED	non-reserved identifier, xml parse
WIDTH	non-reserved identifier, text table column
XMLDECLARATION	non-reserved identifier, xml serialize

Reserved Keywords For Future Use

ALLOCATE	ARE	ASENSITIVE
ASYMETRIC	AUTHORIZATION	BINARY
CALLED	CASCADED	CHARACTER
CHECK	CLOSE	COLLATE
CONNECT	CORRESPONDING	CRITERIA
CURRENT_DATE	CURRENT_TIME	CURRENT_TIMESTAMP
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	RANGE
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
WITHIN	XMLBINARY	XMLDOCUMENT

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
		alter child options list, alter options list, ARRAY expression constructor, column list, create procedure, typed element list, create table body, create temporary table, derived column list, sql exception, named parameter list, expression list, from clause, function, Create GRANT, limit clause, nested expression, object table, option

comma	","	clause, options clause, order by clause, simple data type, query expression, query primary, querystring function, identifier list, Revoke GRANT, select clause, set clause list, in predicate, text aggreate function, text table, xml attributes, xml element, xml query, xml forest, xml namespaces, xml query, xml table
concat_op	"||"	common value expression
decimal numeric literal	(<digit>)* <period> <unsigned integer literal>	unsigned numeric literal
digit	\["0"\-"9"\]	
dollar	"$"	parameter reference
double_amp_op	"&&"	common value expression
eq	"="	assignment statement, callable statement, declare statement, named parameter list, comparison operator, set clause list
escaped function	"{" "fn"	unsigned value expression primary
escaped join	"{" "oj"	table reference
escaped type	"{" ("d" | "t" | "ts" | "b")	non numeric literal
approximate numeric literal	<digit> <period> <unsigned integer literal> \["e","E"\] (<plus> | <minus>)? <unsigned integer literal>	unsigned numeric literal
ge	">="	comparison operator
gt	">"	named parameter list, comparison operator
hexit	\["a"\-"f","A"\-"F"\] | <digit>	
identifier	<quoted_id> (<period> <quoted_id>)*	identifier, unsigned value expression primary
id_part	("@" | "#" | <letter>) (<letter> | "_" | <digit>)*	
lbrace	"{"	callable statement, match predicate
le	"<="	comparison operator
letter	\["a"\-"z","A"\-"Z"\] | \["\u0153"\-"\ufffd"\]	

lparen	"("	standard aggregate function, alter child options list, alter options list, analytic aggregate function, array table, callable statement, column list, other constraints, create procedure, create table body, create temporary table, filter clause, function, group by clause, if statement, Import foreign schema, json object, loop statement, make dep options, nested expression, object table, options clause, ordered aggregate function, simple data type, query primary, querystring function, in predicate, call statement, subquery, quantified comparison predicate, table subquery, table primary, text aggreate function, text table, unescapedFunction, while statement, window specification, with list element, xml attributes, xml element, xml query, xml forest, xml namespaces, xml parse, xml query, xml serialize, xml table
lsbrace	"["	ARRAY expression constructor, data type, value expression primary
lt	"<"	comparison operator
minus	"-"	plus or minus
ne	"<>"	comparison operator
ne2	"!="	comparison operator
period	"."	
plus	"+"	plus or minus
qmark	"?"	callable statement, parameter reference
quoted_id	<id_part> | "\"" ("\"\"" | ~\["\""\])+ "\""	
rbrace	"}"	callable statement, match predicate, non numeric literal, table reference, unsigned value expression primary
rparen	")"	standard aggregate function, alter child options list, alter options list, analytic aggregate function, array table, callable statement, column list, other constraints, create procedure, create table body, create temporary table, filter clause, function, group by clause, if statement, Import foreign schema, json object, loop statement, make dep options, nested expression, object table, options clause, ordered aggregate function, simple data type, query primary, querystring function,

		in predicate, call statement, subquery, quantified comparison predicate, table subquery, table primary, text aggreate function, text table, unescapedFunction, while statement, window specification, with list element, xml attributes, xml element, xml query, xml forest, xml namespaces, xml parse, xml query, xml serialize, xml table
rsbrace	"]"	ARRAY expression constructor, 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
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
ALTER SERVER	alterStatement
alterStatement	ddl statement
ALTER TABLE	alterStatement
ALTER DATA WRAPPER	alterStatement
ALTER TRIGGER	alterStatement
analytic aggregate function	unescapedFunction
ARRAY expression constructor	unsigned value expression primary
array table	table primary
assignment statement	delimited statement
assignment statement operand	assignment statement, declare statement
between predicate	boolean primary
boolean primary	filter clause, boolean factor
branching statement	delimited statement
case expression	unsigned value expression primary
character	match predicate, text aggregate function, text table
column list	other constraints, create temporary table, foreign key, insert statement, primary key, with list element
common value expression	between predicate, boolean primary, comparison predicate, sql exception, 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	create table body

table element	ADD column, create table body
create procedure	ddl statement
create data wrapper aka translator	ddl statement
create database	ddl statement
typed element list	array table, dynamic data statement
create foreign temp table	directly executable statement
option namespace	ddl statement
create role	ddl statement
create schema	ddl statement
create server, aka data source	ddl statement
create table	ddl statement
create table body	create foreign temp table, create table
create temporary table	directly executable statement
create trigger	ddl statement, directly executable statement
condition	expression, having clause, if statement, qualified table, searched case expression, where clause, while statement
cross join	joined table
ddl statement	ddl statement
declare statement	delimited statement
delete statement	assignment statement operand, directly executable statement
delimited statement	statement
derived column	derived column list, object table, querystring function, text aggreate function, xml attributes, xml query, xml query, xml table
derived column list	json object, xml forest
drop option	alter child options list
Drop data wrapper aka translator	ddl statement

drop database	ddl statement
drop option	alter options list
drop procedure	ddl statement
drop role	ddl statement
drop schema	ddl statement
drop server, aka data source	ddl statement
drop table	directly executable statement
drop table	ddl statement
dynamic data statement	data statement
raise error statement	delimited statement
sql exception	assignment statement operand, exception reference
exception reference	sql exception, raise statement
named parameter list	callable statement, call statement
exists predicate	boolean primary
expression	standard aggregate function, ARRAY expression constructor, assignment statement operand, case expression, table element, derived column, dynamic data statement, raise error statement, named parameter list, expression list, function, nested expression, object table column, ordered aggregate function, 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	create table body
from clause	query
function	unescapedFunction, unsigned value expression primary
Create GRANT	ddl statement

group by clause	query
having clause	query
identifier	alter, alter child option pair, alter column options, ALTER DATABASE, DROP column, alter option pair, ALTER PROCEDURE, rename column options, ALTER SERVER, ALTER TABLE, ALTER DATA WRAPPER, ALTER TRIGGER, array table, assignment statement, branching statement, callable statement, column list, compound statement, table element, create procedure, create data wrapper aka translator, create database, typed element list, create foreign temp table, option namespace, create role, create schema, create server, aka data source, create table, create table body, create temporary table, create trigger, declare statement, delete statement, derived column, drop option, Drop data wrapper aka translator, drop database, drop option, drop procedure, drop role, drop schema, drop server, aka data source, drop table, drop table, dynamic data statement, exception reference, named parameter list, foreign key, function, Create GRANT, if statement, Import another Database, Import foreign schema, insert statement, into clause, loop statement, xml namespace element, object table column, object table, option clause, option pair, procedure parameter, procedure result column, query primary, identifier list, Revoke GRANT, select derived column, set clause list, statement, call statement, table subquery, temporary table element, text aggreate function, text table column, text table, table name, update statement, use database, use schema, with list element, xml table column, xml element, xml serialize, xml table
if statement	statement
Import another Database	ddl statement
Import foreign schema	ddl statement
insert statement	assignment statement operand, directly executable statement
integer parameter	fetch clause, limit clause
unsigned integer	dynamic data statement, Create GRANT, integer parameter, make dep options, parameter reference, simple data type, Revoke GRANT, text table column, text table
time interval	function
into clause	query
is null predicate	boolean primary
joined table	table primary, table reference
json object	function
limit clause	query expression body

loop statement	statement
make dep options	option clause, table primary
match predicate	boolean primary
xml namespace element	xml namespaces
nested expression	unsigned value expression primary
non numeric literal	alter child option pair, alter option pair, option pair, value expression primary
non-reserved identifier	identifier, 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	table element, create procedure, create data wrapper aka translator, create database, create schema, create server, aka data source, create table, create table body, Import foreign schema, procedure parameter, procedure result column
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
data type	alter column options, table element, create procedure, typed element list, declare statement, function, object table column, procedure parameter, procedure result column, temporary table element, text table column, unescapedFunction, xml table column
simple data type	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
primary key	create table body

procedure parameter	create procedure
procedure result column	create procedure
qualified table	joined table
query	query primary
query expression	alter, ALTER TABLE, assignment statement operand, create table, insert statement, loop statement, subquery, table subquery, directly executable statement, with list element
query expression body	query expression, query primary
query primary	query term
querystring function	function
query term	query expression body
raise statement	delimited statement
identifier list	create schema, Import foreign schema, with role
grant type	Create GRANT, Revoke GRANT
with role	create role
like regex predicate	boolean primary
return statement	delimited statement
Revoke GRANT	ddl statement
searched case expression	unsigned value expression primary
select clause	query
select derived column	select sublist
select sublist	select clause
set clause list	dynamic data statement, update statement
in predicate	boolean primary
sort key	sort specification
sort specification	order by clause
data statement	delimited statement

statement	alter, ALTER PROCEDURE, compound statement, create procedure, for each row trigger action, if statement, loop statement, procedure body definition, while statement
call statement	assignment statement, subquery, table subquery, directly executable statement
string	character, create database, option namespace, create server, aka data source, drop database, function, Create GRANT, Import another Database, xml namespace element, non numeric literal, object table column, object table, Revoke GRANT, 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
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
unsigned numeric literal	alter child option pair, alter option pair, option pair, value expression primary
unsigned value expression primary	integer parameter, value expression primary
update statement	assignment statement operand, directly executable statement
use database	ddl statement
use schema	ddl statement

directly executable statement	data statement
value expression primary	array table, term
where clause	delete statement, query, update statement
while statement	statement
window specification	unescapedFunction
with list element	query expression
xml attributes	xml element
xml table column	xml table
xml element	function
xml query	boolean primary
xml forest	function
xml namespaces	xml element, xml query, xml forest, xml query, xml table
xml parse	function
xml query	function
xml serialize	function
xml table	table primary

Productions

string ::=

- <string literal>

A string literal value. Use '' to escape ' in the string.

Example:

```
'a string'
```

```
'it''s a string'
```

---

non-reserved identifier ::=

- INSTEAD

- VIEW
- ENABLED
- DISABLED
- KEY
- SERIAL
- TEXTAGG
- COUNT
- ROW_NUMBER
- RANK
- DENSE_RANK
- SUM
- AVG
- MIN
- MAX
- EVERY
- STDDEV_POP
- STDDEV_SAMP
- VAR_SAMP
- VAR_POP
- DOCUMENT
- CONTENT
- TRIM
- EMPTY
- ORDINALITY
- PATH
- FIRST
- LAST
- NEXT
- SUBSTRING
- EXTRACT
- TO_CHARS
- TO_BYTES
- TIMESTAMPADD
- TIMESTAMPDIFF

- QUERYSTRING
- NAMESPACE
- RESULT
- INDEX
- ACCESSPATTERN
- AUTO_INCREMENT
- WELLFORMED
- SQL_TSI_FRAC_SECOND
- SQL_TSI_SECOND
- SQL_TSI_MINUTE
- SQL_TSI_HOUR
- SQL_TSI_DAY
- SQL_TSI_WEEK
- SQL_TSI_MONTH
- SQL_TSI_QUARTER
- SQL_TSI_YEAR
- TEXTTABLE
- ARRAYTABLE
- SELECTOR
- SKIP
- WIDTH
- PASSING
- NAME
- ENCODING
- COLUMNS
- DELIMITER
- QUOTE
- HEADER
- NULLS
- OBJECTTABLE
- VERSION
- INCLUDING
- EXCLUDING
- XMLDECLARATION

- VARIADIC
- RAISE
- EXCEPTION
- CHAIN
- JSONARRAY_AGG
- JSONOBJECT
- PRESERVE
- UPSERT
- AFTER
- TYPE
- TRANSLATOR
- JAAS
- CONDITION
- MASK
- ACCESS
- CONTROL
- NONE
- DATA
- DATABASE
- PRIVILEGES
- ROLE
- SCHEMA
- USE
- REPOSITORY
- RENAME

Allows non-reserved keywords to be parsed as identifiers

Example: SELECT **COUNT** FROM …

---

identifier ::=

- <identifier>
- <non-reserved identifier>

Partial or full name of a single entity.

Example:

```
tbl.col
```

```
"tbl"."col"
```

---

create trigger ::=

- CREATE TRIGGER (<identifier>)? ON <identifier> ((INSTEAD OF) | AFTER) (INSERT | UPDATE | DELETE) AS <for each row trigger action>

Creates a trigger action on the given target.

Example:

```
CREATE TRIGGER ON vw INSTEAD OF INSERT AS FOR EACH ROW BEGIN ATOMIC ... END
```

---

alter ::=

- ALTER ((VIEW <identifier> AS <query expression>) | (PROCEDURE <identifier> AS <statement>) | (TRIGGER (<identifier>)? ON <identifier> ((INSTEAD OF) | AFTER) (INSERT | UPDATE | DELETE) ((AS <for each row trigger action>) | ENABLED | DISABLED)))

Alter the given target.

Example:

```
ALTER VIEW vw AS SELECT col FROM tbl
```

---

for each row trigger action ::=

- FOR EACH ROW ((BEGIN (ATOMIC)? (<statement>)* END) | <statement>)

Defines an action to perform on each row.

Example:

```
FOR EACH ROW BEGIN ATOMIC ... END
```

---

directly executable statement ::=

- <query expression>
- <call statement>
- <insert statement>
- <update statement>
- <delete statement>

- <drop table>
- <create temporary table>
- <create foreign temp table>
- <alter>
- <create trigger>
- <compound statement>

A statement that can be executed at runtime.

Example:

```
SELECT * FROM tbl
```

---

drop table ::=

- DROP TABLE <identifier>

Drop the given table.

Example:

```
DROP TABLE #temp
```

---

create temporary table ::=

- CREATE (LOCAL)? TEMPORARY TABLE <identifier> <lparen> <temporary table element> (<comma> <temporary table element>)* (<comma> PRIMARY KEY <column list>)? <rparen> (ON COMMIT PRESERVE ROWS)?

Creates a temporary table.

Example:

```
CREATE LOCAL TEMPORARY TABLE tmp (col integer)
```

---

temporary table element ::=

- <identifier> (<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> ((<identifier> IS (NOT)? DISTINCT FROM <identifier>) | <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 assigments except for expression are deprecated.{note}

---

data statement ::=

- (<directly executable statement> | <dynamic data statement>) ((WITH | WITHOUT) RETURN)?

A procedure statement that executes a SQL statement. An update statement can have its update count accessed via the ROWCOUNT variable.

---

procedure body definition ::=

- (CREATE (VIRTUAL)? PROCEDURE)? <statement>

Defines a procedure body on a Procedure metadata object.

Example:

```
BEGIN ... END
```

---

dynamic data statement ::=

- (EXECUTE | EXEC) (STRING | IMMEDIATE)? <expression> (AS <typed element list> (INTO <identifier>)?)? (USING <set clause list>)? (UPDATE (<unsigned integer> | <star>))?

A procedure statement that can execute arbitrary sql.

Example:

```
EXECUTE IMMEDIATE 'SELECT * FROM tbl' AS x STRING INTO #temp
```

---

set clause list ::=

- <identifier> <eq> <expression> (<comma> <identifier> <eq> <expression>)*

A list of value assignments.

Example:

```
col1 = 'x', col2 = 'y' ...
```

---

typed element list ::=

- <identifier> <data type> (<comma> <identifier> <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> 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> (<where clause>)? (<option clause>)?

Delete rows from the given target.

Example:

```
DELETE FROM tbl WHERE col2 = 1
```

query expression ::=

- (WITH <with list element> (<comma> <with list element>)*)? <query expression body>

A declarative query for data.

Example:

```
SELECT * FROM tbl WHERE col2 = 1
```

with list element ::=

- <identifier> (<column list>)? AS <lparen> <query expression> <rparen>

A query expression for use in the enclosing query.

Example:

```
X (Y, Z) AS (SELECT 1, 2)
```

---

query expression body ::=

- <query term> ((UNION | EXCEPT) (ALL | DISTINCT)? <query term>)* (<order by clause>)? (<limit clause>)? (<option clause>)?

The body of a query expression, which can optionally be ordered and limited.

Example:

```
SELECT * FROM tbl ORDER BY col1 LIMIT 1
```

---

query term ::=

- <query primary> (INTERSECT (ALL | DISTINCT)? <query primary>)*

Used to establish INTERSECT precedence.

Example:

```
SELECT * FROM tbl
```

```
SELECT * FROM tbl1 INTERSECT SELECT * FROM tbl2
```

---

query primary ::=

- <query>
- (VALUES <lparen> <expression list> <rparen> (<comma> <lparen> <expression list> <rparen>)*)
- (TABLE <identifier>)
- (<lparen> <query expression body> <rparen>)

A declarative source of rows.

Example:

```
TABLE tbl
```

```
SELECT * FROM tbl1
```

---

query ::=

- <select clause> (<into clause>)? (<from clause> (<where clause>)? (<group by clause>)? (<having clause>)?)?

A SELECT query.

Example:

```
SELECT col1, max(col2) FROM tbl GROUP BY col1
```

---

into clause ::=

- INTO <identifier>

Used to direct the query into a table. {note}This is deprecated. Use INSERT INTO with a query expression instead.{note}

Example:

```
INTO tbl
```

---

select clause ::=

- SELECT (ALL | DISTINCT)? (<star> | (<select sublist> (<comma> <select sublist>)*))

The columns returned by a query. Can optionally be distinct.

Example:

```
SELECT *
```

```
SELECT DISTINCT a, b, c
```

---

select sublist ::=

- <select derived column>
- <all in group>

An element in the select clause

Example:

```
tbl.*
```

```
tbl.col AS x
```

---

select derived column ::=

- (<expression> ((AS)? <identifier>)?)

A select clause item that selects a single column. {note}This is slightly different than a derived column in that the AS keyword is optional.{note}

Example:

```
tbl.col AS x
```

---

derived column ::=

- (<expression> (AS <identifier>)?)

An optionally named expression.

Example:

```
tbl.col AS x
```

---

all in group ::=

- <all in group identifier>

A select sublist that can select all columns from the given group.

Example:

```
tbl.*
```

---

ordered aggregate function ::=

- (XMLAGG | ARRAY_AGG | JSONARRAY_AGG) <lparen> <expression> (<order by clause>)? <rparen>

An aggregate function that can optionally be ordered.

Example:

```
XMLAGG(col1) ORDER BY col2
```

```
ARRAY_AGG(col1)
```

---

text aggreate function ::=

- TEXTAGG <lparen> (FOR)? <derived column> (<comma> <derived column>)* (DELIMITER <character>)? ((QUOTE <character>) | (NO QUOTE))? (HEADER)? (ENCODING <identifier>)? (<order by clause>)? <rparen>

An aggregate function for creating separated value clobs.

Example:

```
TEXTAGG (col1 as t1, col2 as t2 DELIMITER ',' HEADER)
```

---

standard aggregate function ::=

- (COUNT <lparen> <star> <rparen>)
- ((COUNT | 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) <lparen> <rparen>

An analytic aggregate function.

Example:

```
ROW_NUMBER()
```

---

filter clause ::=

- FILTER <lparen> WHERE <boolean primary> <rparen>

An aggregate filter clause applied prior to accumulating the value.

Example:

```
FILTER (WHERE col1='a')
```

---

from clause ::=

- FROM (<table reference> (<comma> <table reference>)*)

A query from clause containing a list of table references.

Example:

```
FROM a, b
```

```
FROM a right outer join b, c, d join e".</p>
```

---

table reference ::=

- (<escaped join> <joined table> <rbrace>)
- <joined table>

An optionally escaped joined table.

Example:

```
a
```

```
a inner join b
```

joined table ::=

- <table primary> (<cross join> | <qualified table>)*

A table or join.

Example:

```
a
```

```
a inner join b
```

cross join ::=

- ((CROSS | UNION) JOIN <table primary>)

A cross join.

Example:

```
a CROSS JOIN b
```

qualified table ::=

- (((RIGHT (OUTER)?) | (LEFT (OUTER)?) | (FULL (OUTER)?) | INNER)? JOIN <table reference> ON <condition>)

An INNER or OUTER join.

Example:

```
a inner join b
```

table primary ::=

- (<text table> | <array table> | <xml table> | <object table> | <table name> | <table subquery> | (<lparen> <joined table> <rparen>)) ((MAKEDEP <make dep options>) | MAKENOTDEP)? ((MAKEIND <make dep options>))?

A single source of rows.

Example:

```
a
```

---

make dep options ::=

- (<lparen> (MAX <colon> <unsigned integer>)? ((NO)? JOIN)? <rparen>)?

options for the make dep hint

Example:

```
(min:10000)
```

---

xml serialize ::=

- XMLSERIALIZE <lparen> (DOCUMENT | CONTENT)? <expression> (AS (STRING | VARCHAR | CLOB | VARBINARY | BLOB))? (ENCODING <identifier>)? (VERSION <string>)? ((INCLUDING | EXCLUDING) XMLDECLARATION)? <rparen>

Serializes an XML value.

Example:

```
XMLSERIALIZE(col1 AS CLOB)
```

---

array table ::=

- ARRAYTABLE <lparen> <value expression primary> COLUMNS <typed element list> <rparen> (AS)? <identifier>

The ARRAYTABLE table function creates tabular results from arrays. It can be used as a nested table reference.

Example:

```
ARRAYTABLE (col1 COLUMNS x STRING) AS y
```

---

text table ::=

- TEXTTABLE <lparen> <common value expression> (SELECTOR <string>)? COLUMNS <text table column> (<comma> <text table column>)* ((NO ROW DELIMITER) | (ROW DELIMITER <character>))? (DELIMITER <character>)? ((ESCAPE <character>) | (QUOTE <character>))? (HEADER (<unsigned integer>)?)? (SKIP <unsigned integer>)? (NO TRIM)? <rparen> (AS)? <identifier>

The TEXTTABLE table function creates tabular results from text. It can be used as a nested table reference.

Example:

```
TEXTTABLE (file COLUMNS x STRING) AS y
```

---

text table column ::=

- <identifier> ((FOR ORDINALITY) | ((HEADER <string>)? <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> <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) | (<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>)?)
- <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:

```
=
```

---

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> (ROW | ROWS) (<fetch clause>)?)
- <fetch clause>

Limits and/or offsets the resultant rows.

Example:

```
LIMIT 2
```

---

fetch clause ::=

- FETCH (FIRST | NEXT) (<integer parameter>)? (ROW | ROWS) ONLY

ANSI limit.

Example:

```
FETCH FIRST 1 ROWS ONLY
```

---

option clause ::=

- OPTION (MAKEDEP <identifier> <make dep options> (<comma> <identifier> <make dep options>)* | MAKEIND <identifier> <make dep options> (<comma> <identifier> <make dep options>)* | MAKENOTDEP <identifier> (<comma> <identifier>)* | NOCACHE (<identifier> (<comma> <identifier>)*)?)*

Specifies query options.

Example:

```
OPTION MAKEDEP tbl
```

---

expression ::=

- <condition>

A value.

Example:

```
col1
```

---

common value expression ::=

- (<numeric value expression> ((<double_amp_op> | <concat_op>) <numeric value expression>)*)

Establishes the precedence of concat.

Example:

```
'a' || 'b'
```

---

numeric value expression ::=

- (<term> (<plus or minus> <term>)*)

Example:

```
1 + 2
```

---

plus or minus ::=

- <plus>
- <minus>

The + or - operator.

Example:

```
+
```

---

term ::=

- (<value expression primary> (<star or slash> <value expression primary>)*)

A numeric term

Example:

```
1 * 2
```

---

star or slash ::=

- <star>
- <slash>

The * or / operator.

Example:

```
/
```

---

value expression primary ::=

- <non numeric literal>

- (<plus or minus>)? (<unsigned numeric literal> | (<unsigned value expression primary> (<lsbrace> <numeric value expression> <rsbrace>)*))

A simple value expression.

Example:

```
+col1
```

---

parameter reference ::=

- <qmark>
- (<dollar> <unsigned integer>)

A parameter reference to be bound later.

Example:

```
?
```

---

unescapedFunction ::=

- ((<text aggreate function> | <standard aggregate function> | <ordered aggregate function>) (<filter clause>)? (<window specification>)?) | (<analytic aggregate function> (<filter clause>)? <window specification>) | (<function> (<window specification>)?)
- (XMLCAST <lparen> <expression> AS <data type> <rparen>)

---

nested expression ::=

- (<lparen> (<expression> (<comma> <expression>)*)? (<comma>)? <rparen>)

An expression nested in parens

Example:

```
(1)
```

---

unsigned value expression primary ::=

- <parameter reference>
- (<escaped function> <function> <rbrace>)
- <unescapedFunction>
- <identifier> | <non-reserved identifier>
- <subquery>
- <nested expression>

- <ARRAY expression constructor>
- <searched case expression>
- <case expression>

An unsigned simple value expression.

Example:

```
col1
```

ARRAY expression constructor ::=

- (ARRAY <lsbrace> (<expression> (<comma> <expression>)*)? <rsbrace>)

Creates and array of the given expressions.

Example:

```
----ARRAY[1,2]
----
```

---

window specification ::=

- OVER <lparen> (PARTITION BY <expression list>)? (<order by clause>)? <rparen>

The window specification for an analytical or windowed aggregate function.

Example:

```
OVER (PARTION BY col1)
```

---

case expression ::=

- CASE <expression> (WHEN <expression> THEN <expression>)+ (ELSE <expression>)? END

If/then/else chain using a common search predicand.

Example:

```
CASE col1 WHEN 'a' THEN 1 ELSE 2
```

---

searched case expression ::=

- CASE (WHEN <condition> THEN <expression>)+ (ELSE <expression>)? END

If/then/else chain using multiple search conditions.

Example:

```
CASE WHEN x = 'a' THEN 1 WHEN y = 'b' THEN 2
```

function ::=

- (CONVERT <lparen> <expression> <comma> <data type> <rparen>)
- (CAST <lparen> <expression> AS <data type> <rparen>)
- (SUBSTRING <lparen> <expression> ((FROM <expression> (FOR <expression>)?) | (<comma> <expression list>)) <rparen>)
- (EXTRACT <lparen> (YEAR | MONTH | DAY | HOUR | MINUTE | SECOND) FROM <expression> <rparen>)
- (TRIM <lparen> ((((LEADING | TRAILING | BOTH) (<expression>)?) | <expression>) FROM)? <expression> <rparen>)
- ((TO_CHARS | TO_BYTES) <lparen> <expression> <comma> <string> (<comma> <expression>)? <rparen>)
- ((TIMESTAMPADD | TIMESTAMPDIFF) <lparen> <time interval> <comma> <expression> <comma> <expression> <rparen>)
- <querystring function>
- ((LEFT | RIGHT | CHAR | USER | YEAR | MONTH | HOUR | MINUTE | SECOND | XMLCONCAT | XMLCOMMENT | XMLTEXT) <lparen> (<expression list>)? <rparen>)
- ((TRANSLATE | INSERT) <lparen> (<expression list>)? <rparen>)
- <xml parse>
- <xml element>
- (XMLPI <lparen> ((NAME)? <identifier>) (<comma> <expression>)? <rparen>)
- <xml forest>
- <json object>
- <xml serialize>
- <xml query>
- (<identifier> <lparen> (ALL | DISTINCT)? (<expression list>)? (<order by clause>)? <rparen> (<filter clause>)?)

Calls a scalar function.

Example:

```
func('1', col1)
```

xml parse ::=

- XMLPARSE <lparen> (DOCUMENT | CONTENT) <expression> (WELLFORMED)? <rparen>

Parses the given value as XML.

Example:

```
XMLPARSE(DOCUMENT doc WELLFORMED)
```

querystring function ::=

- QUERYSTRING <lparen> <expression> (<comma> <derived column>)* <rparen>

Produces a URL query string from the given arguments.

Example:

```
QUERYSTRING('path', col1 AS opt, col2 AS val)
```

xml element ::=

- XMLELEMENT <lparen> ((NAME)? <identifier>) (<comma> <xml namespaces>)? (<comma> <xml attributes>)? (<comma> <expression>)* <rparen>

Creates an XML element.

Example:

```
XMLELEMENT(NAME "root", child)
```

xml attributes ::=

- XMLATTRIBUTES <lparen> <derived column> (<comma> <derived column>)* <rparen>

Creates attributes for the containing element.

Example:

```
XMLATTRIBUTES(col1 AS attr1, col2 AS attr2)
```

json object ::=

- JSONOBJECT <lparen> <derived column list> <rparen>

Produces a JSON object containing name value pairs.

Example:

```
JSONOBJECT(col1 AS val1, col2 AS val2)
```

derived column list ::=

- <derived column> (<comma> <derived column>)*

a list of name value pairs

Example:

```
col1 AS val1, col2 AS val2
```

---

xml forest ::=

- XMLFOREST <lparen> (<xml namespaces> <comma>)? <derived column list> <rparen>

Produces an element for each derived column.

Example:

```
XMLFOREST(col1 AS ELEM1, col2 AS ELEM2)
```

---

xml namespaces ::=

- XMLNAMESPACES <lparen> <xml namespace element> (<comma> <xml namespace element>)* <rparen>

Defines XML namespace URI/prefix combinations

Example:

```
XMLNAMESPACES('http://foo' AS foo)
```

---

xml namespace element ::=

- (<string> AS <identifier>)
- (NO DEFAULT)
- (DEFAULT <string>)

An xml namespace

Example:

```
NO DEFAULT
```

---

simple data type ::=

- (STRING (<lparen> <unsigned integer> <rparen>)?)
- (VARCHAR (<lparen> <unsigned integer> <rparen>)?)
- BOOLEAN
- BYTE
- TINYINT
- SHORT
- SMALLINT

- (CHAR (<lparen> <unsigned integer> <rparen>)?)
- INTEGER
- LONG
- BIGINT
- (BIGINTEGER (<lparen> <unsigned integer> <rparen>)?)
- FLOAT
- REAL
- DOUBLE
- (BIGDECIMAL (<lparen> <unsigned integer> (<comma> <unsigned integer>)? <rparen>)?)
- (DECIMAL (<lparen> <unsigned integer> (<comma> <unsigned integer>)? <rparen>)?)
- DATE
- TIME
- TIMESTAMP
- (OBJECT (<lparen> <unsigned integer> <rparen>)?)
- (BLOB (<lparen> <unsigned integer> <rparen>)?)
- (CLOB (<lparen> <unsigned integer> <rparen>)?)
- (VARBINARY (<lparen> <unsigned integer> <rparen>)?)
- GEOMETRY
- XML

A non-collection data type.

Example:

```
STRING
```

---

data type ::=

- <simple data type> (<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 server, aka data source>
- <create role>
- <drop role>
- <Create GRANT>
- <Revoke GRANT>
- <drop server, aka data source>
- <drop table>
- <Import foreign schema>
- <Import another Database>
- <create database>
- <drop database>
- <use database>
- <drop schema>
- <use schema>
- <create schema>
- <create procedure> (<ddl statement>)?
- <create data wrapper aka translator>
- <Drop data wrapper aka translator>
- <drop procedure>

A data definition statement.

Example:

```
CREATE FOREIGN TABLE X (Y STRING)
```

---

option namespace ::=

- SET NAMESPACE <string> AS <identifier>

A namespace used to shorten the full name of an option key.

Example:

```
SET NAMESPACE 'http://foo' AS foo
```

create database ::=

- CREATE DATABASE <identifier> (VERSION <string>)? (<options clause>)?

create a new database

Example:

```
CREATE DATABASE foo [VERSION 'version'] OPTIONS(...)
```

use database ::=

- USE DATABASE <identifier> (VERSION <string>)?

database into working context

Example:

```
USE DATABASE foo [VERSION 'version']
```

drop database ::=

- DROP DATABASE <identifier> (VERSION <string>)?

drop database

Example:

```
DROP DATABASE foo [VERSION 'version']
```

create schema ::=

- CREATE (VIRTUAL)? SCHEMA <identifier> (SERVER <identifier list>)? (<options clause>)?

create a schema in database

Example:

```
CREATE [VIRTUAL] SCHEMA foo SERVER (s1,s2,s3) OPTIONS(...)
```

drop schema ::=

- DROP (VIRTUAL)? SCHEMA <identifier>

drop a schema in database

Example:

```
----DROP SCHEMA foo
----
```

---

use schema ::=

- SET SCHEMA <identifier>

use schema for following database resources

Example:

```
USE SCHEMA foo
```

---

create data wrapper aka translator ::=

- CREATE FOREIGN (DATA WRAPPER | TRANSLATOR) <identifier> (TYPE <identifier>)? (<options clause>)?

Defines a translator; use the options to override the translator properties.

Example:

```
CREATE FOREIGN (DATA WRAPPER|TRANSLATOR)  wrapper OPTIONS(properties)
```

Drop data wrapper aka translator ::=

- DROP FOREIGN (DATA WRAPPER | TRANSLATOR) <identifier>

Deletes a translator

Example:

```
DROP FOREIGN (DATA WRAPPER|TRANSLATOR) wrapper
```

---

create role ::=

- CREATE ROLE <identifier> (WITH <with role>)?

Defines data role for the database

Example:

```
CREATE DATA ROLE <data-role>  [WITH JAAS ROLE <string>(,<string>)*]
```

---

with role ::=

- (JAAS ROLE <identifier list> | ANY AUTHENTICATED) (WITH (JAAS ROLE <identifier list> | ANY AUTHENTICATED))*

---

drop role ::=

- DROP ROLE <identifier>

Removes data role for the database

Example:

```
DROP ROLE <data-role>
```

Create GRANT ::=

- GRANT <grant type> (<comma> <grant type>)* ON (DATABASE (<identifier>)? | TABLE <identifier> (CONDITION (CONSTRAINT)? <string>)? | PROCEDURE <identifier> (CONDITION (CONSTRAINT)? <string>)? | SCHEMA <identifier> | COLUMN <identifier> (MASK (ORDER <unsigned integer>)? <string>)? (CONDITION (CONSTRAINT)? <string>)?) TO <identifier>

Defines GRANT for a role

Example:

```
GRANT ( SELECT | INSERT |  UPDATE | DELETE | EXECUTE | LANGUAGE | ALTER | DROP | ALL PRIVILEGES | TEMPORARY TAB
LES )

'''


=== [[revokeGrantOption]]_<<usage_revokeGrantOption, Revoke GRANT>>_ ::=

* <<token_REVOKE,REVOKE>> <<token_GRANT,GRANT>> <<<readGrantTypes,grant type>>> ( <<<token_COMMA,comma>>> <<<re
adGrantTypes,grant type>>> )* <<token_ON,ON>> ( <<token_DATABASE,DATABASE>> ( <<<id,identifier>>> )? | <<token_
TABLE,TABLE>> <<<id,identifier>>> ( <<token_CONDITION,CONDITION>> ( <<token_CONSTRAINT,CONSTRAINT>> )? <<<strin
gVal,string>>> )? | <<token_PROCEDURE,PROCEDURE>> <<<id,identifier>>> ( <<token_CONDITION,CONDITION>> ( <<token
_CONSTRAINT,CONSTRAINT>> )? <<<stringVal,string>>> )? | <<token_SCHEMA,SCHEMA>> <<<id,identifier>>> | <<token_C
OLUMN,COLUMN>> <<<id,identifier>>> ( <<token_MASK,MASK>> ( <<token_ORDER,ORDER>> <<<intVal,unsigned integer>>>
)? <<<stringVal,string>>> )? ( <<token_CONDITION,CONDITION>> ( <<token_CONSTRAINT,CONSTRAINT>> )? <<<stringVal,
string>>> )? ) <<token_FROM,FROM>> <<<id,identifier>>>


Revokes GRANT for a role


Example:
[source,sql]
```

REVOKE GRANT (SELECT | INSERT | UPDATE | DELETE | EXECUTE | LANGUAGE | ALTER | DROP | ALL PRIVILEGES | TEMPORARY TABLES)

---

create server, aka data source ::=

- CREATE SERVER <identifier> (TYPE <string>)? (VERSION <string>)? FOREIGN (DATA WRAPPER | TRANSLATOR) <identifier> (<options clause>)?

Defines connection to foreign source

Example:

```
CREATE SERVER server_name [ TYPE 'server_type' ] [ VERSION 'server_version' ] FOREIGN (<DATA> <WRAPPER>|<TRANSL
ATOR>)  fdw_name [ OPTIONS ( option 'value' [, ... ] ) ]
```

---

drop server, aka data source ::=

- DROP SERVER <identifier>

Defines dropping connection to foreign source

Example:

```
----DROP SERVER server_name
----
```

---

create procedure ::=

- CREATE (VIRTUAL | FOREIGN)? (PROCEDURE | FUNCTION) <identifier> (<lparen> (<procedure parameter> (<comma> <procedure parameter>)*)? <rparen> (RETURNS (<options clause>)? (((TABLE)? <lparen> <procedure result column> (<comma> <procedure result column>)* <rparen>) | <data type>))? (<options clause>)? (AS <statement>)?)

Defines a procedure or function invocation.

Example:

```
CREATE FOREIGN PROCEDURE proc (param STRING) RETURNS STRING
```

---

drop procedure ::=

- DROP (VIRTUAL | FOREIGN)? (PROCEDURE | FUNCTION) <identifier>

Drops a table or view.

Example:

```
DROP [FOREIGN (TABLE|VIEW) table-name
```

---

procedure parameter ::=

- (IN | OUT | INOUT | VARIADIC)? <identifier> <data type> (NOT NULL)? (RESULT)? (DEFAULT <expression>)? (<options clause>)?

A procedure or function parameter

Example:

```
OUT x INTEGER
```

---

procedure result column ::=

- <identifier> <data type> (NOT NULL)? (<options clause>)?

A procedure result column.

Example:

```
x INTEGER
```

---

create table ::=

- CREATE ((FOREIGN TABLE) | ((VIRTUAL)? VIEW) | (GLOBAL TEMPORARY TABLE)) <identifier> (<create table body> | (<options clause>)?) (AS <query expression>)?

Defines a table or view.

Example:

```
CREATE VIEW vw AS SELECT 1
```

---

drop table ::=

- DROP ((FOREIGN TABLE) | ((VIRTUAL)? VIEW) | (GLOBAL TEMPORARY TABLE)) <identifier>

Drops a table or view.

Example:

```
DROP (FOREIGN TABLE | [VIRTUAL] VIEW) table-name
```

---

create foreign temp table ::=

- CREATE (LOCAL)? FOREIGN TEMPORARY TABLE <identifier> <create table body> ON <identifier>

Defines a foreign temp table

Example:

```
CREATE FOREIGN TEMPORARY TABLE t (x string) ON z
```

---

create table body ::=

- <lparen> <table element> (<comma> <table element>)* (<comma> (CONSTRAINT <identifier>)? (<primary key> | <other constraints> | <foreign key>) (<options clause>)?)* <rparen> (<options clause>)?

Defines a table.

Example:

```
(x string) OPTIONS (CARDINALITY 100)
```

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)?)) ((PRIMARY KEY) | ((UNIQUE)? (INDEX)?)) (DEFAULT <expression>)? (<options clause>)?

Defines a table column.

Example:

```
x INTEGER NOT NULL
```

---

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 (VIRTUAL | FOREIGN)? (<ALTER TABLE> | <ALTER PROCEDURE> | <ALTER TRIGGER> | <ALTER SERVER> | <ALTER DATA WRAPPER> | <ALTER DATABASE>)

ALTER TABLE ::=

- (VIEW <identifier> AS <query expression> (<alter options list> | <alter column options>)?)
- (TABLE <identifier> (<alter options list> | <ADD column> | <DROP column> | <alter column options> | <rename column options>)?)

alters options of database

Example:

```
ALTER TABLE foo (ADD|DROP|ALTER) COLUMN <name> <type> OPTIONS ( (ADD|SET|DROP) x y)
```

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 ::=

- ALTER (COLUMN | PARAMETER) <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 ::=

- RENAME (COLUMN | PARAMETER) <identifier> TO <identifier>

renames either a table column or procedure's parameter name

Example:

```
RENAME COLUMN bar TO foo
```

ALTER PROCEDURE ::=

- PROCEDURE <identifier> (AS <statement>)? (<alter options list> | <alter column options> | <rename column options>)?

alters options of database

Example:

```
ALTER PROCEDURE foo [AS <stmt>] OPTIONS (ADD x y)
```

ALTER TRIGGER ::=

- TRIGGER ON <identifier> INSTEAD OF (INSERT | UPDATE | DELETE) (AS <for each row trigger action> | ENABLED | DISABLED)

alters options of table triggers

Example:

```
ALTER TRIGGER ON <id> INSTEAD OF (INSERT|UPDATE|DELETE) AS [ENABLED|DISABLED]
```

ALTER SERVER ::=

- SERVER <identifier> <alter options list>

alters options of database

Example:

```
ALTER SERVER foo OPTIONS (ADD x y)
```

ALTER DATA WRAPPER ::=

- (DATA WRAPPER | TRANSLATOR) <identifier> <alter options list>

alters options of data wrapper

Example:

```
ALTER [DATA WRAPPER|TRANSLATOR] foo OPTIONS (ADD x y)
```

ALTER DATABASE ::=

- DATABASE <identifier> <alter options list>

alters options of database

Example:

```
ALTER DATABASE foo OPTIONS (ADD x y)
```

alter options list ::=

- OPTIONS <lparen> (<add set option> | <drop option>) (<comma> (<add set option> | <drop option>))* <rparen>

a list of alterations to options

Example:

```
OPTIONS (ADD updatable true)
```

drop option ::=

- DROP <identifier>

drop option

Example:

```
DROP updatable
```

add set option ::=

- (ADD | SET) <alter option pair>

add or set an option pair

Example:

```
ADD updatable true
```

alter child options list ::=

- OPTIONS <lparen> (<add set child option> | <drop option>) (<comma> (<add set child option> | <drop option>))* <rparen>

a list of alterations to options

Example:

```
OPTIONS (ADD updatable true)
```

drop option ::=

- DROP <identifier>

drop option

Example:

```
DROP updatable
```

---

add set child option ::=

- (ADD | SET) <alter child option pair>

add or set an option pair

Example:

```
ADD updatable true
```

---

alter child option pair ::=

- <identifier> (<non numeric literal> | (<plus or minus>)? <unsigned numeric literal>)

Alter An option key/value pair.

Example:

```
'key' 'value'
```

Import foreign schema ::=

- IMPORT FOREIGN SCHEMA <identifier> (LIMIT TO <lparen> <identifier list> <rparen> | <lparen> EXCEPT <identifier list> <rparen>)? FROM (SERVER | REPOSITORY) <identifier> INTO <identifier> (<options clause>)?

imports schema metadata from server

Example:

```
IMPORT FOREIGN SCHEMA foo [LIMIT TO (x,y,z)|EXCEPT (x,y,z)] FROM SERVER bar
```

Import another Database ::=

- IMPORT DATABASE <identifier> VERSION <string> (WITH ACCESS CONTROL)?

imports another database into current database

Example:

```
IMPORT DATABASE <id> VERSION <string-val> [WITH ACCESS CONTROL]
```

---

identifier list ::=

- <identifier> (<comma> <identifier>)*

---

grant type ::=

- SELECT
- INSERT
- UPDATE
- DELETE
- EXECUTE
- LANGUAGE
- ALTER
- DROP
- ALL PRIVILEGES
- TEMPORARY TABLE

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.

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 and adminshell) 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 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.
- "pg" - Defaults to no SSL

Warning	The pg transport 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.

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 AES scheme.
  - For the pg transport 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 pg transport.

SSL Authentication Modes

- **anonymous** – 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. When the client or server lack the anonymous cipher suite, consider using 1-way 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.

- **1-way** – 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.
- **2-way** – 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 expected 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.

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.

1-way

1. server.keystore - has server's private key
2. server.truststore - has server's public key

2-way

1. server.keystore - has server's private key
2. server.truststore - has server's public key
3. client.keystore - client's private key
4. client.truststore - has client's public key

Full Configuration Options

Example XML Configuration

```
<ssl mode="enabled" authentication-mode="1-way" 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 - anonymous|1-way|2-way, Type of SSL 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 if 1-way or 2-way authentication is used.
- 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 "2-way".
- 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 1-way and 2-way 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. You can download the driver from http://www.postgresql.org/ftp/odbc/versions/

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.

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 based on Teiid Transport

User can define a "default" authentication per transport as below that can be used for all the VDBs system wide.

For JDBC:

```
Use below CLI commands to edit the configuration
----
/subsystem=teiid/transport=jdbc:write-attribute(name=authentication-security-domain, value=MY_REALM)
/subsystem=teiid/transport=jdbc:write-attribute(name=authentication-type, value=GSS)
----
```

Will result in following changes (or you can edit the standalone-teiid.xml file directly)

```
<transport name="jdbc" protocol="teiid" socket-binding="teiid-jdbc"/>
        <authentication security-domain="MY_REALM" type="GSS"/>
</transport>
```

For ODBC:

```
Use below CLI commands to edit the configuration
----
/subsystem=teiid/transport=odbc:write-attribute(name=authentication-security-domain, value=MY_REALM)
/subsystem=teiid/transport=odbc:write-attribute(name=authentication-type, value=GSS)
----
```

```
<transport name="odbc" protocol="pg" socket-binding="teiid-odbc"/>
    <authentication security-domain="MY_REALM" type="GSS"/>
</transport>
```

"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 also add these properties inside standalone-teiid.xml file, right after \{<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.jboss.teiid</groupId>
         <artifactId>teiid-api</artifactId>
         <scope>provided</scope>
      </dependency>
      <dependency>
         <groupId>org.jboss.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.jboss.teiid.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.jboss.teiid.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.jboss.teiid.common-core"/>
    <module name="org.jboss.teiid.api"/>
    <module name="javax.api"/>
</dependencies>
</module>
```

create folder in the "<jboss-as>/modules/org/jboss/teiid/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.SAML2L
oginModule, 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:

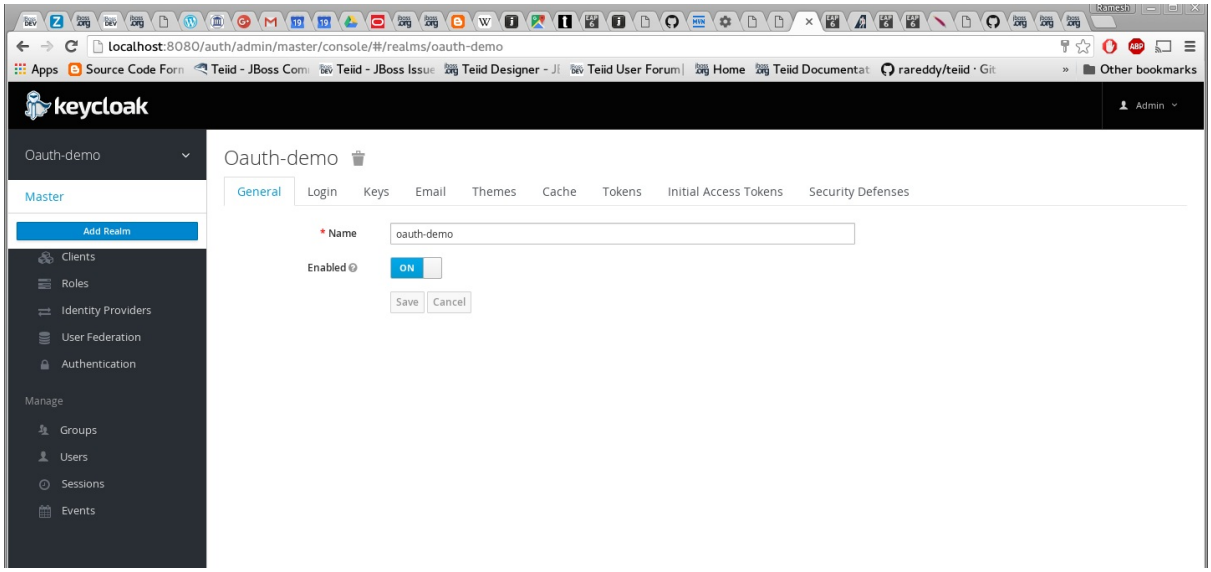
```
deployment-overlay add --name=myOverlay --content=/WEB-INF/web.xml=/modified/web.xml,/WEB-INF/jboss-web.xml=/mo
dified/jboss-web.xml --deployments=teiid-odata-odata4.war --redeploy-affected
```

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 implementaion as OAuth implementions 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.

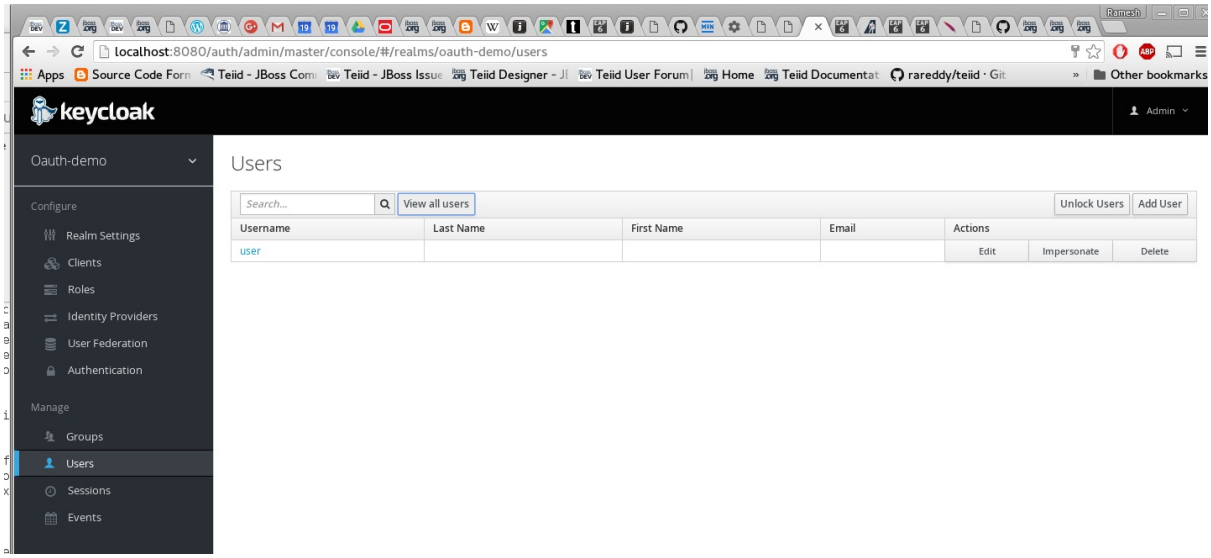
This examples will show case an example, where Teiid's OData rest interface is secured using OAuth using Keycloak as IDP. The VDB accessed by the OData interface also depends on another web service which is used as a data source, that is also secured with OAuth using the same Keycloak IDP. The central 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.

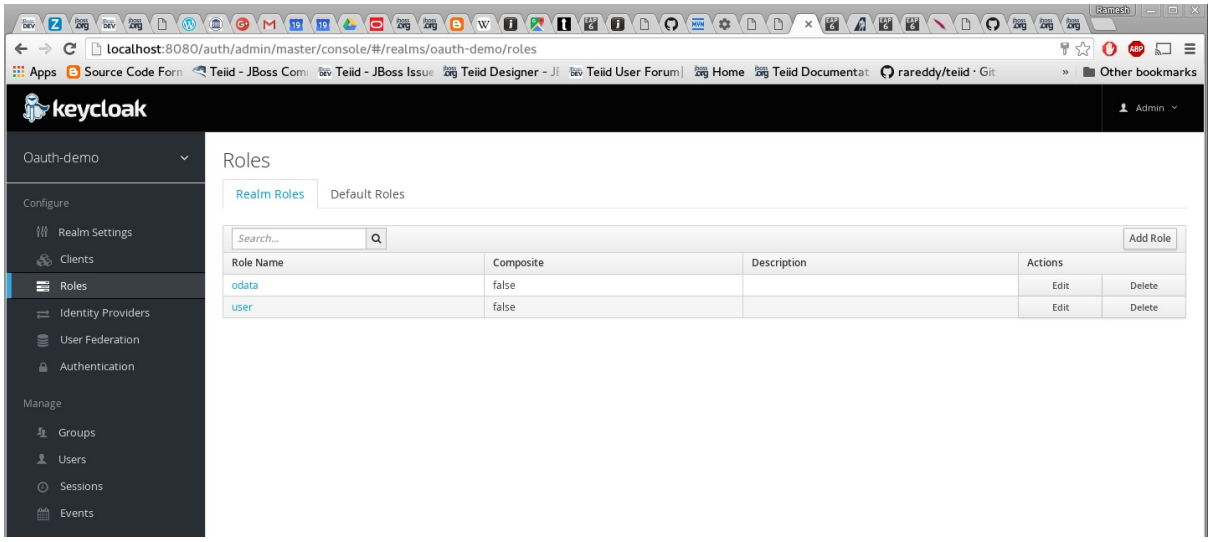
- Login using the default "admin/admin" credentials into the Keycloak "master" realm.
- Add a new realm called "oauth-demo"



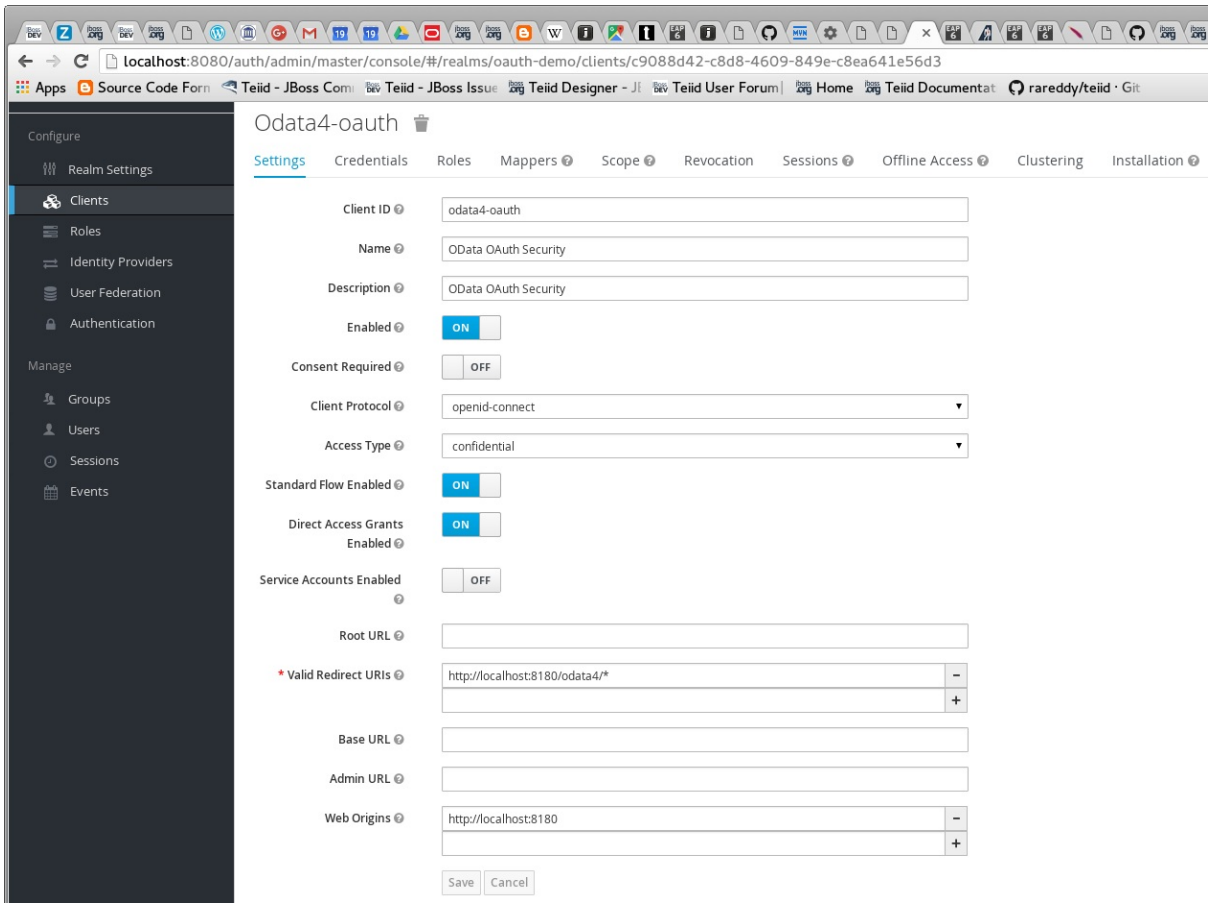
- Add a new user called "user" and add credentials.



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.

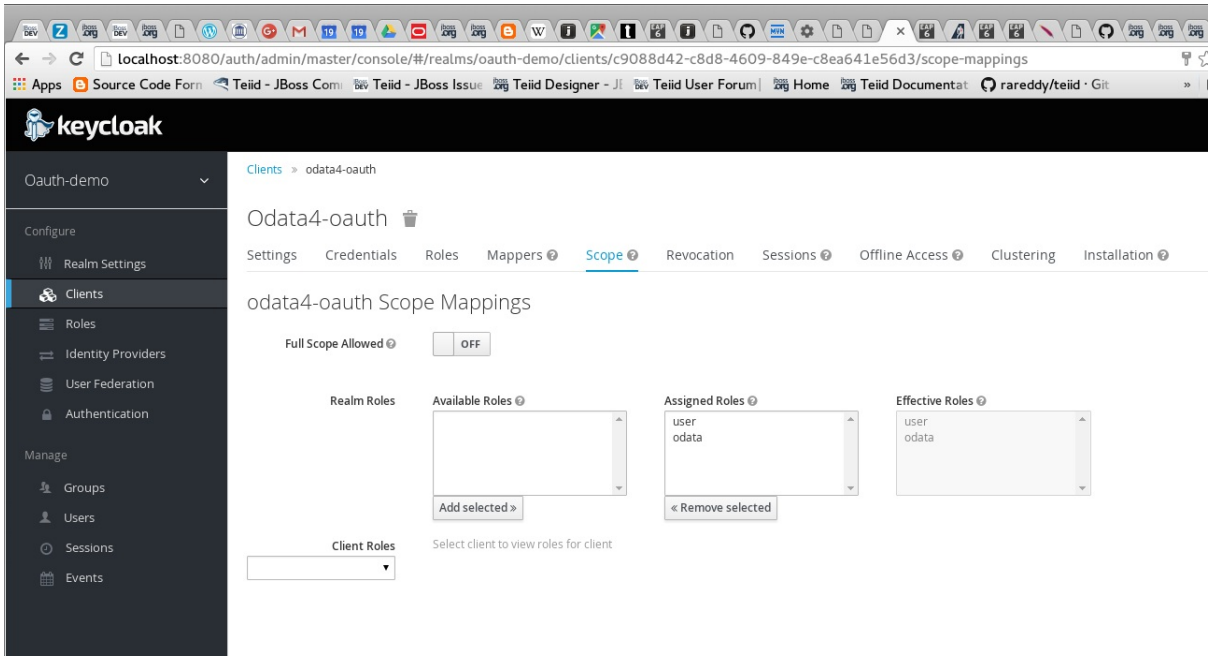


- Add a new client called "odata4-oauth", this client represents the Teiid's OData client that we are going to create

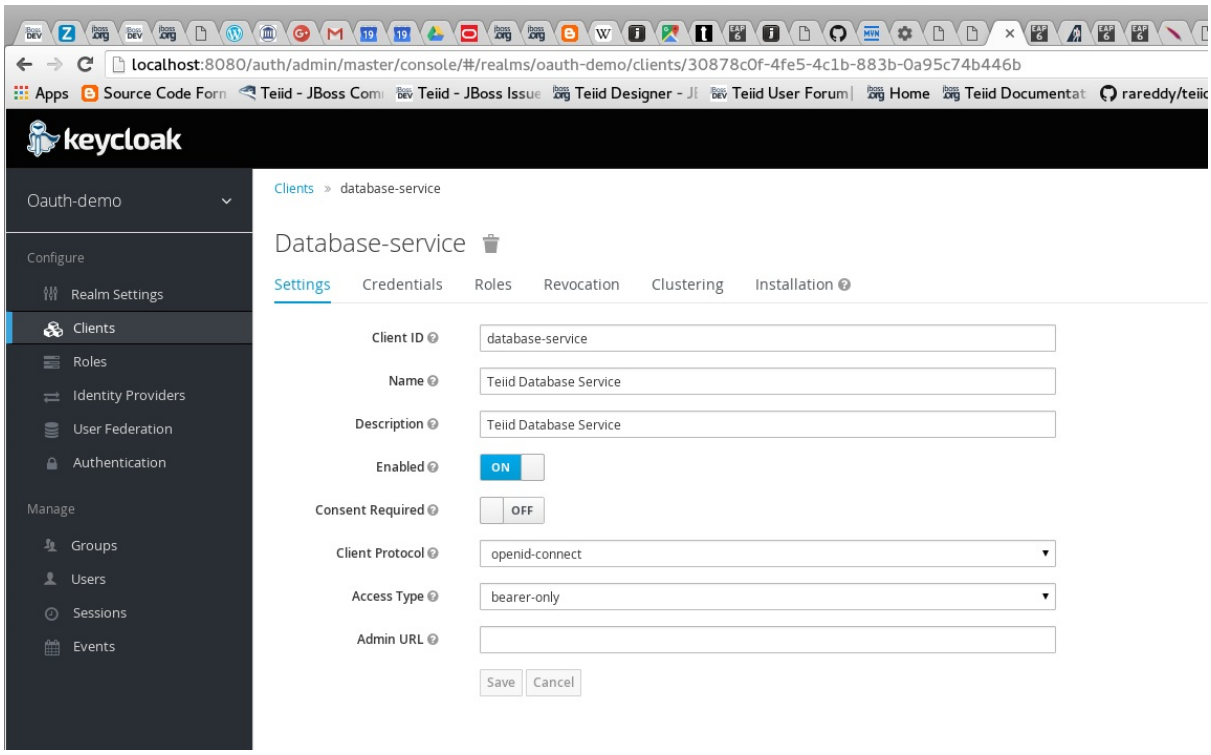


and choose scopes "odata" and "user" for this client. Note that the redirect URI needs to be where the actual service is going to be available.

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
- Download Keycloak SAML adapter for EAP, and unzip over the Teiid server installation. (optional but need for another exercise)
- Edit the standalone-teiid.xml, add the following sections

Run the following CLI to add Keycloak specific modules to the server

```
/extension=org.keycloak.keycloak-saml-adapter-subsystem:add(module=org.keycloak.keycloak-saml-adapter-subsystem
)
/extension=org.keycloak.keycloak-adapter-subsystem:add(module=org.keycloak.keycloak-adapter-subsystem)
```

above commands will result in XML in standalone.xml or domain.xml file like

```
    <extension module="org.keycloak.keycloak-saml-adapter-subsystem"/>
    <extension module="org.keycloak.keycloak-adapter-subsystem"/>
```

Add these two subsystems any where in the file, use the following the CLI script

```
/subsystem=keycloak:add
/subsystem=keycloak-saml:add
```

above commands will result in XML in standalone.xml or domain.xml file like

```
        <subsystem xmlns="urn:jboss:domain:keycloak-saml:1.1"/>
        <subsystem xmlns="urn:jboss:domain:keycloak:1.1"/>
```

In security-domains add following security domains using the following CLI

```
/subsystem=security/security-domain=oauth:add(cache-type=default)
/subsystem=security/security-domain=oauth/authentication=classic:add
/subsystem=security/security-domain=oauth/authentication=classic/login-
module=oauth:add(code=org.teiid.jboss.PassthroughIdentityLoginModule,
flag=required, module=org.jboss.teiid)

/subsystem=security/security-domain=keycloak:add(cache-type=default)
/subsystem=security/security-domain=keycloak/authentication=classic:add
/subsystem=security/security-domain=keycloak/authentication=classic/login-
module=keycloak:add(code=org.keycloak.adapters.jboss.KeycloakLoginModule,
flag=required)

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="oauth">
        <authentication>
            <login-module code="org.teiid.jboss.PassthroughIdentityLoginModule" flag="required" module="org.jbo
ss.teiid"/>
        </authentication>
    </security-domain>
    <security-domain name="keycloak">
        <authentication>
            <login-module code="org.keycloak.adapters.jboss.KeycloakLoginModule" flag="required"/>
        </authentication>
    </security-domain>
```

Under Teiid subsystem, change the "security-domain" of the "odata" transport, to

```
/subsystem=teiid/transport=odata:write-attribute(name=authentication-security-domain, value=oauth)
```

results in XML

```
<transport name="odata">
    <authentication security-domain="oauth"/>
</transport>
```

This finishes all the server side changes that are required to make OAuth authentication using Keycloak.

OData Application WAR

In order to use OAuth 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 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.

to build the WAR files running the maven command

```
mvn clean package
```

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

{code} 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 {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-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

Edit the standalone-teiid.xml and under resource-adapters subsystem, add the following to add access to a database-service from the Teiid query engine.

```
    <resource-adapter id="database">
        <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:/database" enabled="true" use-java-context="true"
                                   pool-name="teiid-database-ds">
                <config-property name="SecurityType">
                    OAuth
                </config-property>
                <config-property name="EndPoint">
                    http://localhost:8180/database/
                </config-property>
                <security>
                    <security-domain>oauth</security-domain>
                </security>
            </connection-definition>
        </connection-definitions>
    </resource-adapter>
```

Add a VDB with following contents (oauthdemo-vdb.xml)

```
<vdb name="oauthdemo" version="1">
    <model visible="true" name="PM1">
        <source name="array" translator-name="loopback"/>
        <metadata type = "DDL"><![CDATA[
            CREATE FOREIGN TABLE G1 (e1 integer PRIMARY KEY, e2 varchar(25), e3 double);
        ]]>
       </metadata>
    </model>

    <model name="view" type="VIRTUAL">
        <metadata type = "DDL"><![CDATA[
            create view message (msgto string primary key, msgfrom string, heading string, body string)
            as
                SELECT A.msgto, A.msgfrom, A.heading, A.body
                FROM
                  (EXEC restsvc.invokeHttp(action=>'GET', endpoint=>'sample', stream=>'TRUE')) AS f,
                  XMLTABLE('/note' PASSING XMLPARSE(DOCUMENT f.result) COLUMNS
                    msgto string PATH 'to',
                    msgfrom string PATH 'from',
                    heading string PATH 'heading',
                    body string PATH 'body') AS A;
        ]]>
       </metadata>
    </model>
```

```
    <model name="restsvc" type="PHYSICAL" visible="true">
        <property name="importer.importWSDL" value="false"/>
        <source name="restsvc" translator-name="ws" connection-jndi-name="java:/database"/>
    </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/kerberos/auth
```

then you should see a message like "Congratulations!!! Login successful..". What this process is doing is negotiating a "access-token" from the Keycloak authentication server and places this in the client's web-session, such that subsequent calls to the service use this token for access.

Now to fetch the data from the "database-service" using the negotiated "access-token" issue a query

```
http://localhost:8180/odata4/oauthdemo/view/message
```

If all the configuration is setup correctly, then you will see the response like below.

```
<?xml version='1.0' encoding='UTF-8'?>
<a:feed xmlns:a="http://www.w3.org/2005/Atom" xmlns:m="http://docs.oasis-open.org/odata/ns/metadata"
    xmlns:d="http://docs.oasis-open.org/odata/ns/data" m:context="$metadata#mesage">
    <a:id>http://localhost:8180/odata4/saml.1/RestViewModel/mesage</a:id>
    <a:entry>
        <a:id>mesage('Tove')</a:id>
        <a:title />
        <a:summary />
        <a:updated>2016-01-18T20:10:48Z</a:updated>
        <a:author>
            <a:name />
        </a:author>
        <a:link rel="edit" href="mesage('Tove')" />
        <a:category scheme="http://docs.oasis-open.org/odata/ns/scheme"
            term="#saml.1.RestViewModel.mesage" />
        <a:content type="application/xml">
            <m:properties>
                <d:msgto>Tove</d:msgto>
                <d:msgfrom>Jani</d:msgfrom>
```

```
                <d:heading>Reminder</d:heading>
                <d:body>Don't forget me this weekend!</d:body>
            </m:properties>
        </a:content>
    </a:entry>
</a:feed>
```

Warning	When above method is used to capture access token, it is possible that the access token gets expired after its lifespan, in that situation a new access token needs to be negotiated.

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/view/message
```

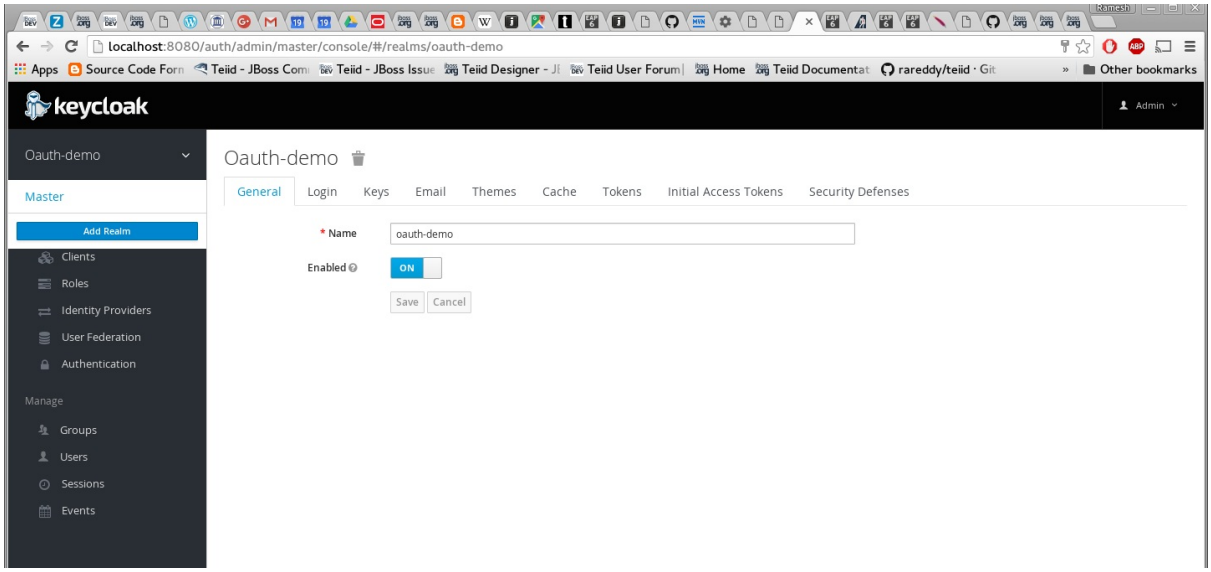
You should see same XML response as above. Please note that to programatically achieve the access_token in your own program (not using curl) you can see some suggestions in this document [http://keycloak.github.io/docs/userguide/keycloak-server/html/direct-access-grants.html]

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.

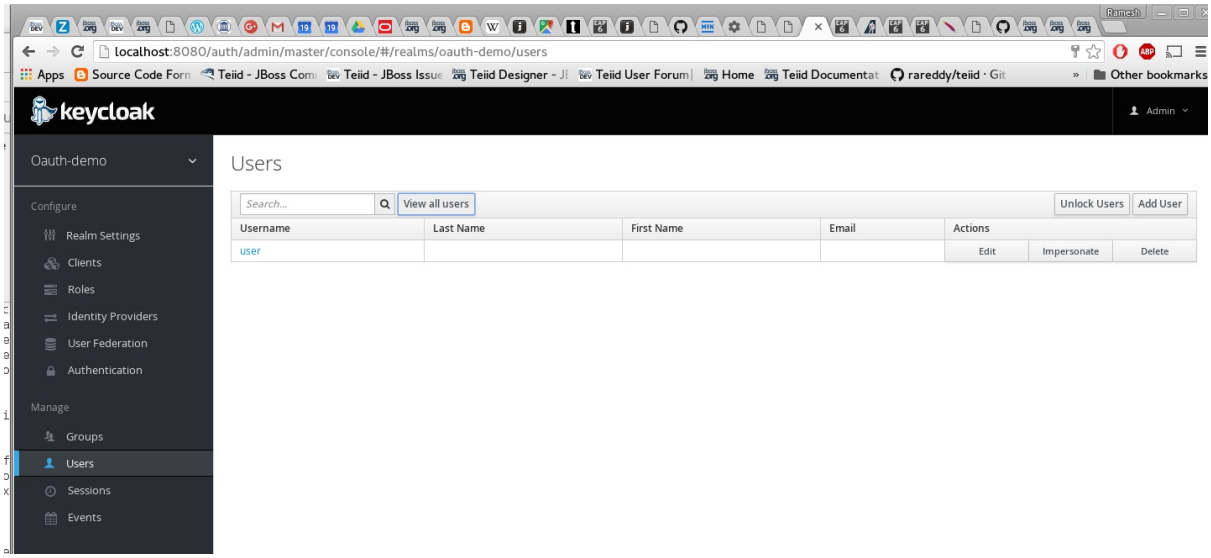
This examples will show case an example, where Teiid's OData rest interface is secured using SAML using Keycloak as IDP. The VDB accessed by the OData interface, the pass-through of SAML Assertion for OAuth token (SAML Bearer) is not yet available in KeyCloak, when the feature is available then Teiid 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)

Download and install Keycloak as a separate web server.

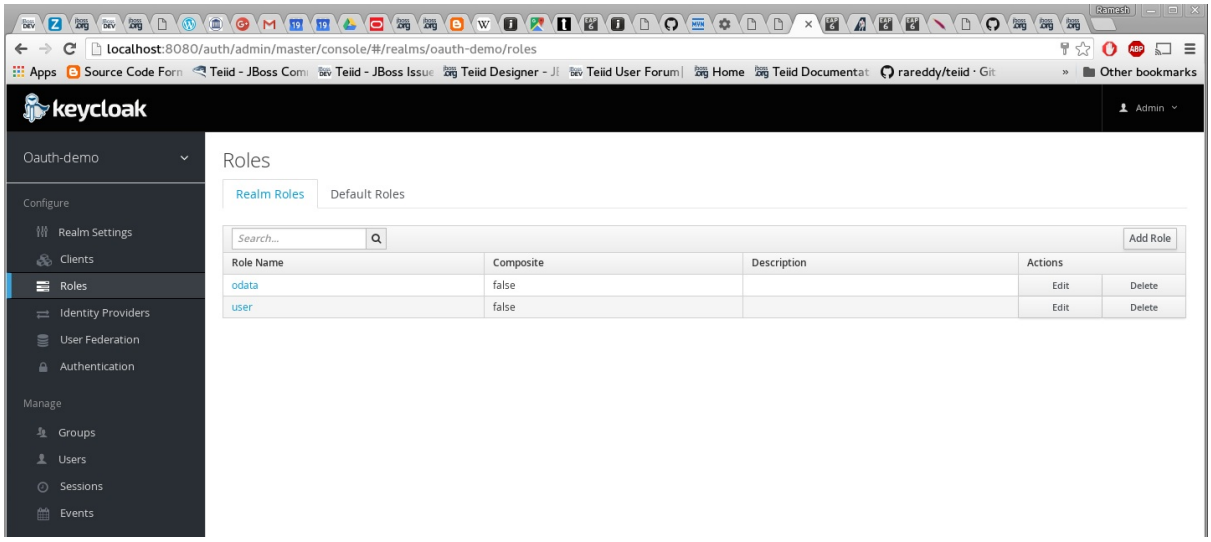
- Login using the default "admin/admin" credentials into the Keycloak "master" realm.
- Add a new realm called "oauth-demo"



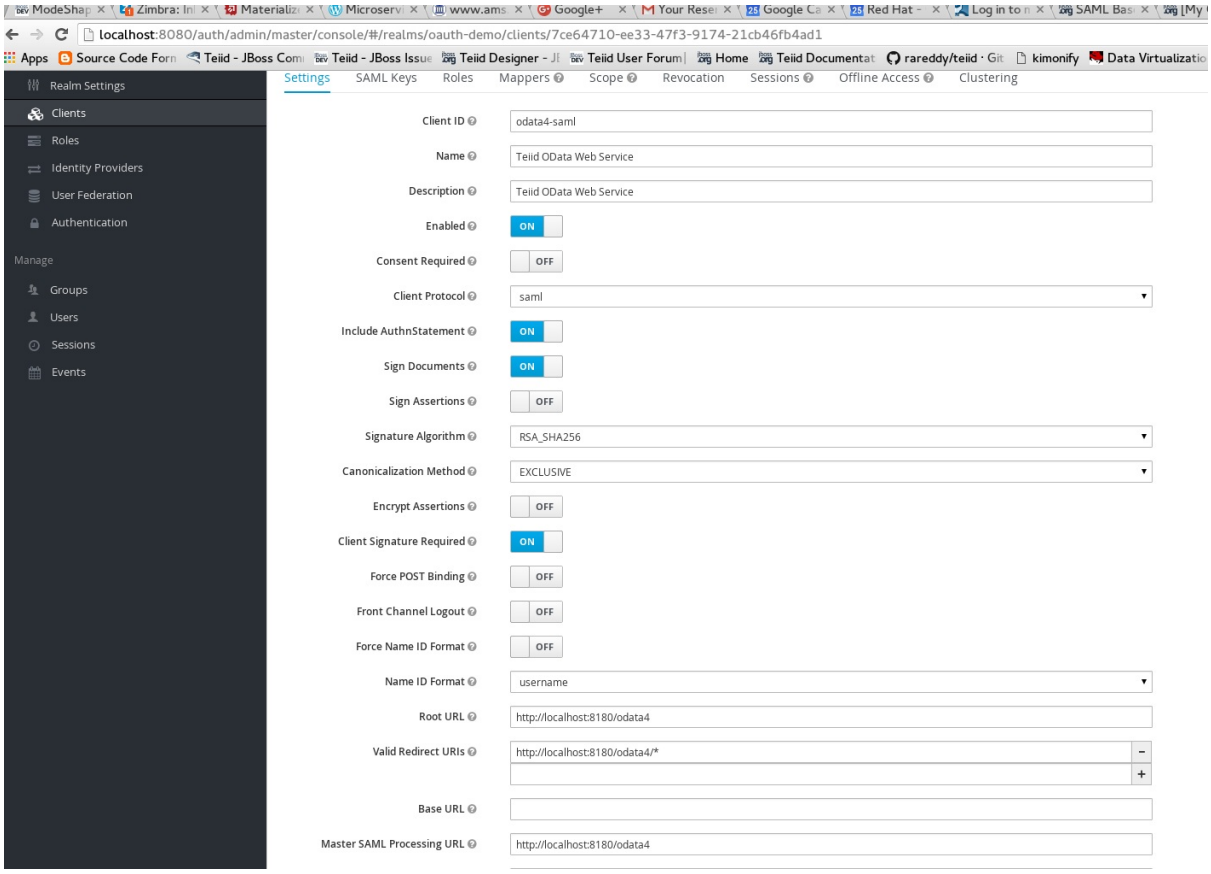
- Add a new user called "user" and add credentials.



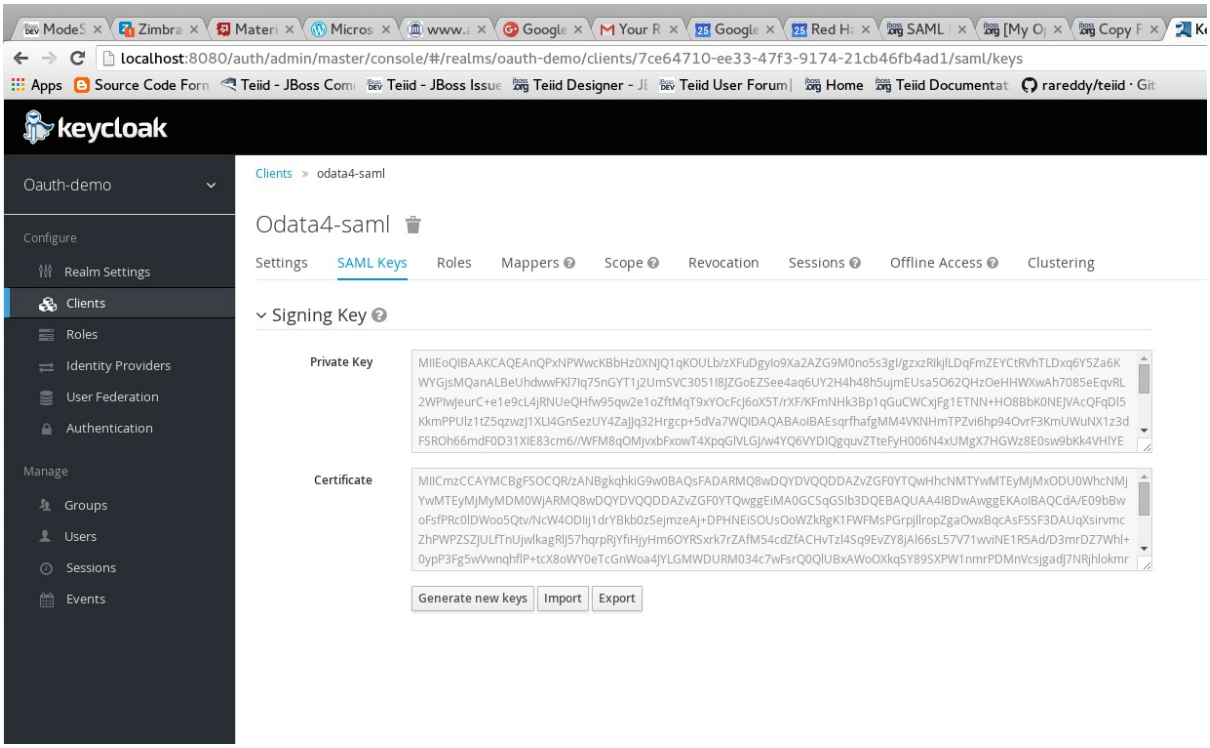
Add two roles "odata" and "user". These are enterprise roles, that will be used by the web services to grant the access to user. Also these roles are used as "scopes" in the OAuth protocol.



- 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 adapter for the EAP, and unzip over the Teiid server installation
- Download Keycloak SAML adapter for EAP, and unzip over the Teiid server installation. (optional but need for another exercise)
- Edit the standalone-teiid.xml, add the following sections

Run the following CLI to add Keycloak specific modules to the server

```
/extension=org.keycloak.keycloak-saml-adapter-
subsystem:add(module=org.keycloak.keycloak-saml-adapter-subsystem)
/extension=org.keycloak.keycloak-adapter-
subsystem:add(module=org.keycloak.keycloak-adapter-subsystem)
```

above commands will result in XML in standalone.xml or domain.xml file like

```
    <extension module="org.keycloak.keycloak-saml-adapter-subsystem"/>
    <extension module="org.keycloak.keycloak-adapter-subsystem"/>
```

Add these two subsystems any where in the file, use the following the CLI script

```
/subsystem=keycloak:add
/subsystem=keycloak-saml:add
```

above commands will result in XML in standalone.xml or domain.xml file like

```
        <subsystem xmlns="urn:jboss:domain:keycloak-saml:1.1"/>
        <subsystem xmlns="urn:jboss:domain:keycloak:1.1"/>
```

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=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="teiid-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 OAuth authentication using Keycloak.

OData Application WAR

In order to use OAuth 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
```

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
```

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>
```