Red Hat Integration 2020.Q1

Developing Clients for Data Virtualization

TECHNOLOGY PREVIEW - Guide for client developers

Last Updated: 2021-02-19

Red Hat Integration 2020.Q1 Developing Clients for Data Virtualization

TECHNOLOGY PREVIEW - Guide for client developers

Legal Notice

Abstract

Learn how to connect to Data Virtualization and access your data through the client interfaces.

Table of Contents

CHAPTER 1. DEVELOPING CLIENTS FOR DATA VIRTUALIZATION

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

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

IMPORTANT

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

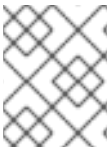
CHAPTER 2. JDBC COMPATIBILITY

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

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

2.1. GENERATED KEYS

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



NOTE

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

2.2. CONNECTING TO A DATA VIRTUALIZATION SERVER

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

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

Once you have the VDB deployed in Data Virtualization, client applications can connect to the Data Virtualization Server and issue SQL queries against deployed VDB using JDBC API. If you are new to JDBC, see Java's documentation about JDBC. Data Virtualization ships with teiid-1.3.0-jdbc.jar that can be found in the downloads.

You can also obtain the Data Virtualization JDBC from the Maven Repository https://oss.sonatype.org/content/repositories/releases/ using the coordinates:

```
<dependency>
  <groupId>org.teiid</groupId>
  <artifactId>teiid</artifactId>
  <classifier>jdbc</classifier>
  <version>1.3.0</version>
</dependency>
```

Important classes in the client JAR:

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

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

2.2.1. OpenTracing compatibility

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

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

where version.opentracing (0.31 for Data Virtualization 11.0) is defined in the project integration bom.

Or you may manually include the opentracing-util, opentracing-api, and opentracing-noop jars as needed by the tooling or other environment where the Data Virtualization client jar is utilized.

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

2.2.2. 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 Data Virtualization JDBC protocol, mms defines a secure channel (see SSL Client Connections for more)
3. <host> - defines the server where the Data Virtualization 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 Data Virtualization Server is listening for incoming JDBC connections.

5. [prop-name=prop-value] - additionally you can supply any number of name value pairs separated by semi-colon [;]. All compatible URL properties are defined in the connection properties section. Property values should be URL encoded if they contain reserved characters, e.g. ('?', '=', ';', etc.)

2.2.2.1. Local Connections

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

2.2.2.2. URL Connection Properties

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

Table 2.1. 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 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 Data Virtualization 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

Property Name	Type	Description
ansiQuotedIdentifiers	**boolean**	Sets the parsing behavior for double quoted entries in SQL. The default, true, parses doubled 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. If true in embedded mode, connections will reconnect to a newer VDB of the same name/version.
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 Data Virtualization looks for already authenticated security context on the calling thread. If one found it uses that users credentials to create session. Data Virtualization 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 Data Virtualization 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.

Property Name	Type	Description
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 with JDBC3 and earlier. Defaults to true.
jaasName	**String**	JAAS configuration name. Only applies when configuring a GSS authentication. Defaults to Data Virtualization. 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 Data Virtualization views as a VIEW table type. If false then Data Virtualization views will be reported as a TABLE. Default true.

2.2.2.3. Client SSL Settings

The following sections define the properties required for each SSL mode. Note that when connecting to Data Virtualization Server with SSL enabled, you *MUST* use the *"mms"* protocol, instead of "mm" in the JDBC connection URL, for example

```
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.

2.2.2.3.1. Option 1: Java SSL properties

These are standard Java defined system properties to configure the SSL under any JVM, Data Virtualization 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)
```

2.2.2.3.2. Option 2: Data Virtualization 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 Data Virtualization-specific certificate keys into those https certificate keystores is not allowed.

In this scenario, a different set of Data Virtualization-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)
```

2.2.3. Additional Socket Client Settings

A "teiid-client-settings.properties" file can be used to configure Data Virtualization 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

#
# Set the max time to live (in milliseconds) for non-execution
# synchronous calls.
#

org.teiid.sockets.synchronousttl=240000

#
# Set the socket receive buffer size (in bytes)
# 0 indicates that the default socket setting will be used.
#

org.teiid.sockets.receiveBufferSize=0

#
# Set the socket send buffer size (in bytes)
# 0 indicates that the default socket setting will be used.
#

org.teiid.sockets.sendBufferSize=0

#
# Set to true to enable Nagle's algorithm to conserve bandwidth
# by minimizing the number of segments that are sent.
#

org.teiid.sockets.conserveBandwidth=false

#
# Maximum number of bytes per server message.
# May need to be increased when using custom types and/or large batch sizes.
#

org.teiid.sockets.maxObjectSize=33554432
```

NOTE

All properties listed in "teiid-client-settings.properties" can also be set as System or env properties.

2.3. PREPARED STATEMENTS

Data Virtualization 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 Data Virtualization **PreparedStatements**, since Data Virtualization 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)).
- If you have the same value of a binding repeated multiple times in your query, you can consolidate that usage in a couple of ways.
  - The query can be enclosed as a anonymous procedure block:

```
BEGIN
 DECLARE string PARAM1 = cast(? as string);
 SELECT ... WHERE COLUMN1 = $1 AND COLUMN2 = $1 ...;
```

Note the cast of the bind variable, which is due to a small issue with the resolver that isn't inferring the type from the variable declaration.

- You can also use the PostgreSQL like feature of $n positional bindings:

```
SELECT ... WHERE COLUMN1 = $1 AND COLUMN2 = $1 ...
```

2.4. RESULTSET LIMITATIONS

The following limitations apply to result sets in Data Virtualization:

- TYPE_SCROLL_SENSITIVE are not compatible.
- **UPDATABLE** ResultSets are not compatible.
- You cannot return multiple ResultSets from a Procedure execution.

2.5. JDBC EXTENSIONS

These are custom extensions to JDBC API from Data Virtualization to provide compatibility with various features.

2.5.1. Statement Extensions

The Data Virtualization 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 2.2. 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.

2.5.2. Partial Results Mode

You can use a "partial results" query mode with the Data Virtualization Server. In this mode, the behavior of the query processor changes so that the server returns results even when some data sources are unavailable.

For example, suppose that two data sources exist for different suppliers and your data designers have created a virtual group that creates a union between the information from the two suppliers. If your application submits a query without using partial results query mode and one of the suppliers' databases

is down, the query against the virtual group returns an exception. However, if your application runs the same query in "partial results" query mode, the server returns data from the running data source and no data from the data source that is down.

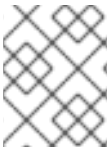
When using "partial results" mode, if a source throws an exception during processing it does not cause the user's query to fail. Rather, that source is treated as returning no more rows after the failure point. Most commonly, that source will return 0 rows.

This behavior is most useful when using **UNION** or **OUTER JOIN** queries as these operations handle missing information in a useful way. Most other kinds of queries will simply return 0 rows to the user when used in partial results mode and the source is unavailable.

For each source that is excluded from the query, a warning will be generated describing the source and the failure. These warnings can be obtained from the **Statement.getWarnings()** method. This method returns a **SQLWarning** object but in the case of "partial results" warnings, this will be an object of type **org.teiid.jdbc.PartialResultsWarning** class. This class can be used to obtain a list of all the failed sources by name and to obtain the specific exception thrown by each source.

NOTE

Because Data Virtualization enables cursoring before an entire result is formed, it is possible that a data source failure will not be determined until after the first batch of results have been returned to the client. This can happen in the case of unions, but not joins. To ensure that all warnings have been accumulated, the statement should be checked after the entire result set has been read.



NOTE

If other warnings are returned by execution, then the partial results warnings may occur after the first warning in the warning chain.

Partial results mode is off by default but can be turned on for all queries in a Connection with either setPartialResultsMode("true") on a DataSource or partialResultsMode=true on a JDBC URL. In either case, partial results mode may be toggled later with a SET Statement.

Setting Partial Results Mode

```
Statement statement = ...obtain statement from Connection...
statement.execute("set partialResultsMode true");
```

Getting Partial Results Warnings

```
statement.execute("set partialResultsMode true");
ResultSet results = statement.executeQuery("SELECT Name FROM Accounts");
while (results.next()) {
  ... //process the result set
}
SQLWarning warning = statement.getWarnings();
while(warning != null) {
  if (warning instanceof PartialResultsWarning) {
    PartialResultsWarning partialWarning = (PartialResultsWarning)warning;
    Collection failedConnectors = partialWarning.getFailedConnectors();
    Iterator iter = failedConnectors.iterator();
    while(iter.hasNext()) {
```

```
      String connectorName = (String) iter.next();
      SQLException connectorException = partialWarning.getConnectorException(connectorName);
      System.out.println(connectorName + ": " + connectorException.getMessage());
    }
  }
  warning = warning.getNextWarning();
}
```

WARNING

In some instances, typically JDBC sources, the source not being initially available will prevent Data Virtualization 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.

2.5.3. 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);
Data VirtualizationPreparedStatement tStmt = stmt.unwrap(Data
VirtualizationPreparedStatement.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 **Data VirtualizationResultSet.available()** to determine if your desired positioning will be valid.

2.5.3.1. 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.

2.5.4. ResultSet Extensions

The Data Virtualization 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 Data Virtualization statement. The following methods are provided on the extension interface:

Table 2.3. 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.

2.5.5. Connection Extensions

Data Virtualization connections (defined by the **org.teiid.jdbc.TeiidConnection** interface) are compatible with the changeUser method to reauthenticate a given connection. If the reauthentication is successful the current connection my be used with the given identity. Existing statements/result sets are still available for use under the old identity.

2.6. INCOMPATIBLE JDBC METHODS

Based upon the JDBC in JDK 1.6, this appendix details only those JDBC methods that Data Virtualization is not compatible with. Unless specified below, Data Virtualization is compatible with all other JDBC Methods.

Those methods listed without comments throw a SQLException stating that it is not supported.

Where specified, some listed methods do not throw an exception, but possibly exhibit unexpected behavior. If no arguments are specified, then all related (overridden) methods are not compatible. If an argument is listed then only those forms of the method specified are not compatible.

2.6.1. Incompatible 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 ----

Class name	Methods
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**

Class name	Methods
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**

2.6.2. Incompatible Classes and Methods in "javax.sql"

Class name	Methods
RowSet*	**Not Supported**

CHAPTER 3. ODBC COMPATIBILITY

Open Database Connectivity (ODBC) is a standard database access method developed by the SQL Access group in 1992. ODBC, just like JDBC in Java, allows consistent client access regardless of which database management system (DBMS) is handling the data. ODBC uses a driver to translate the application's data queries into commands that the DBMS understands. For this to work, both the application and the DBMS must be ODBC-compliant – that is, the application must be capable of issuing ODBC commands and the DBMS must be capable of responding to them.

Data Virtualization can provide ODBC access to deployed VDBs in the Data Virtualization runtime through PostgreSQL's ODBC driver. This is possible because Data Virtualization 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 Data Virtualization queries - not pgsql queries. Although you can use many PostgreSQL constructs, the default behavior for queries matches Data Virtualization's expectations. See System Properties for optional properties that further emulate pgsql handling.

NOTE

Handling names with underscore ("_") in ODBC. By default Data Virtualization does not have a default like escape character. Depending upon the ODBC client however there may be an expectation that backslash is used by default - which is the behavior of PostgreSQL. This may cause metadata queries to be issued against objects with "_" in their name to return no or incorrect results. You may globally emulate the behavior of PostgreSQL by setting the **org.teiid.backslashDefaultMatchEscape** system property to **true**. To alter the property just for the current session then have your ODBC client issue **select cast(teiid_session_set('backslashDefaultMatchEscape', true) as boolean)** statement before any other statement.

Postgres ODBC drivers 9.5 and later do not require this special property as the client will use an E escaped literal instead.

Compatibility was last ensured with the 9.6 Postgres ODBC driver. You are encouraged to use later client versions when needed and report any issues to the community.

3.1. 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 Data Virtualization 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.

3.2. 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 Data Virtualization VDB.

3.3. CONFIGURATION

WARNING

By default, clients use plain text password authentication in Data Virtualization for pg/ODBC interfaces. If the client/server are not configured to use SSL or GSS authentication, the password will be sent in plain text over the network.

For a windows client, see the Configuring the Data Source Name.

See also DSN Less Connection.

3.3.1. Connection Settings

All the available pg driver connection options with their descriptions that can be used are defined here https://odbc.postgresql.org/docs/config.html. When using these properties on the connection string, their property names are defined here https://odbc.postgresql.org/docs/config-opt.html.

However Data Virtualization does not honor all properties, and some, such as Updatable Cursors, will cause query failures.

Table 3.1. Primary ODBC Settings For Data Virtualization

Name	Description
Updateable Cursors & Row Versioning	Should not be used.
Use serverside prepare & Parse Statements & Disallow Premature	It is recommended that "Use serverside prepare" is enabled and "Parse Statements"/"Disallow Premature" are disabled
SSL mode	May be needed if you are connecting to a secured pg transport port.
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 Data Virtualization 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.

3.3.1.1. Data Virtualization Connection Settings

Most Data Virtualization 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.

3.4. CONFIGURING THE DATA SOURCE NAME (DSN)

See Data Virtualization compatible options for a description of the available client configuration options.

3.4.1. Windows Installation

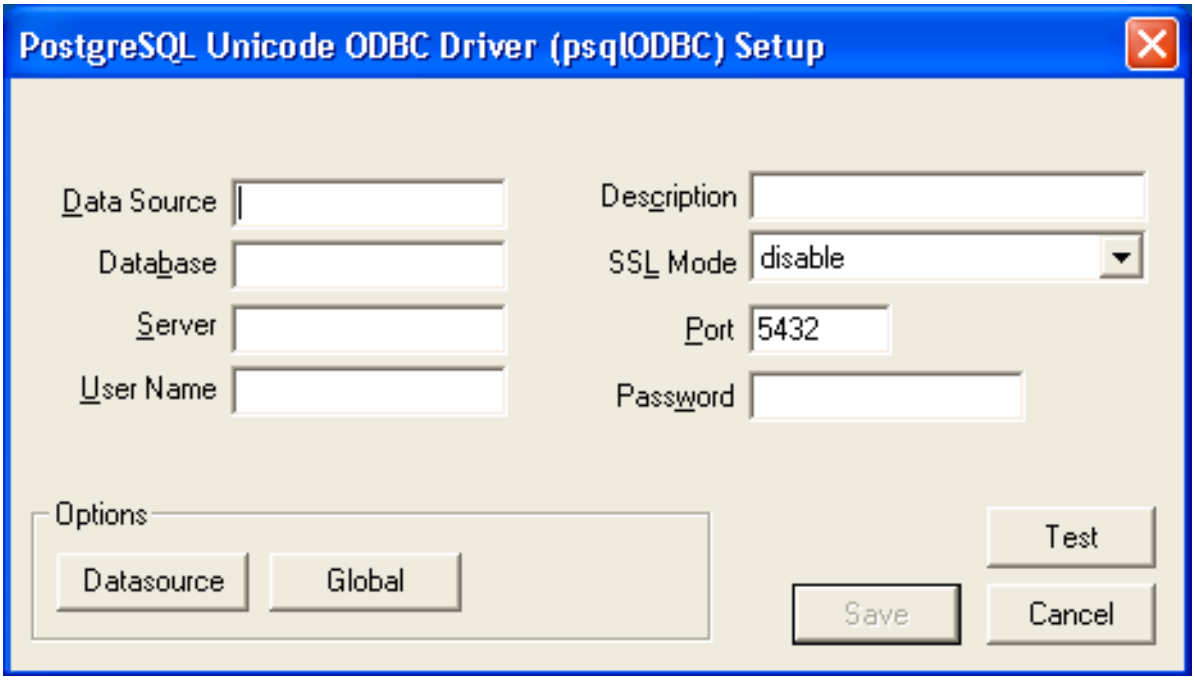
Once you have installed the ODBC Driver Client software on your workstation, you have to configure it to connect to a Data Virtualization Runtime. Note that the following instructions are specific to the Microsoft Windows Platform.

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 Data Virtualization System. To make more than one VDB available, you need to configure more than one data source name.

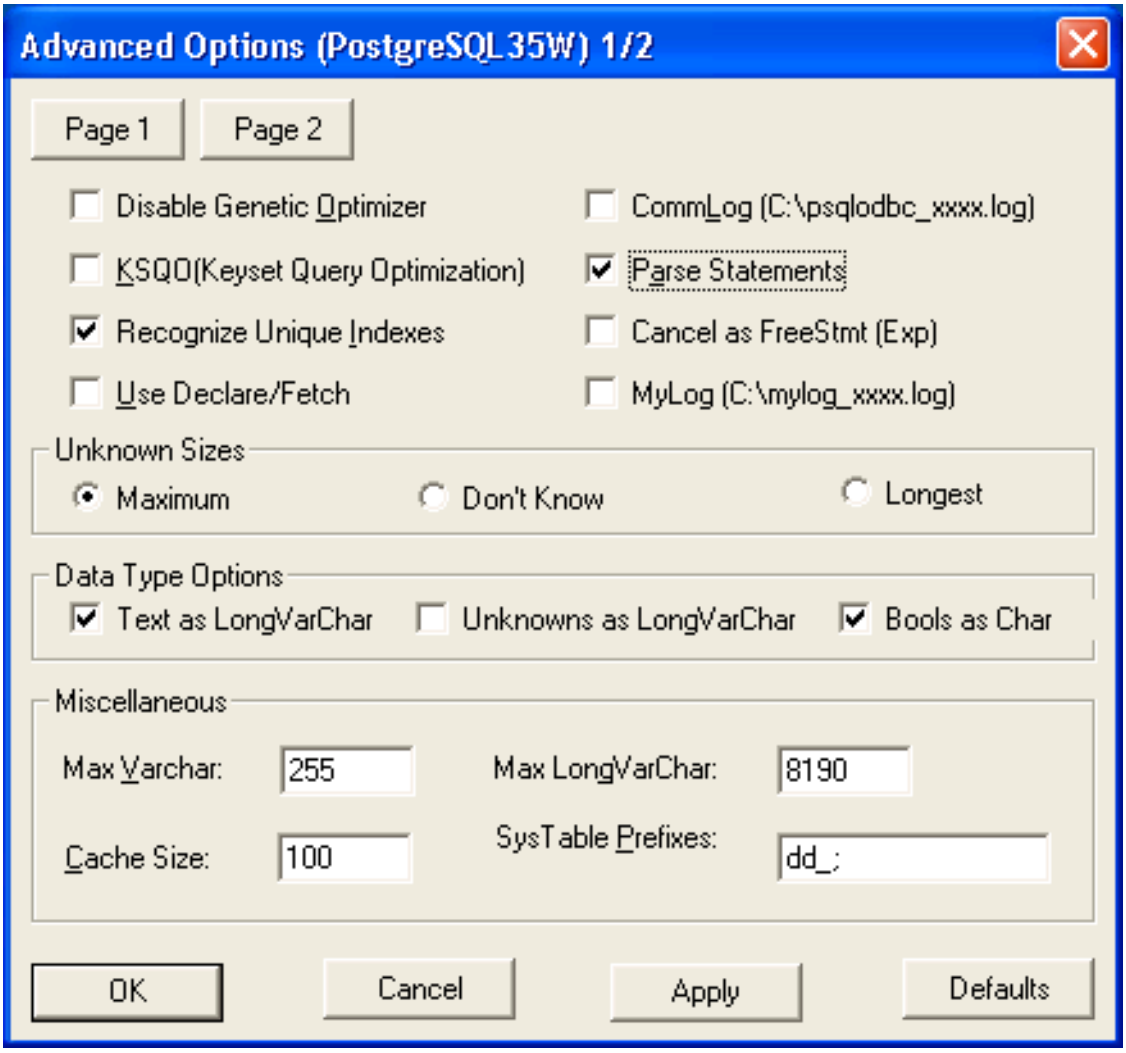
Follow the below steps in creating a data source name (DSN)

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.

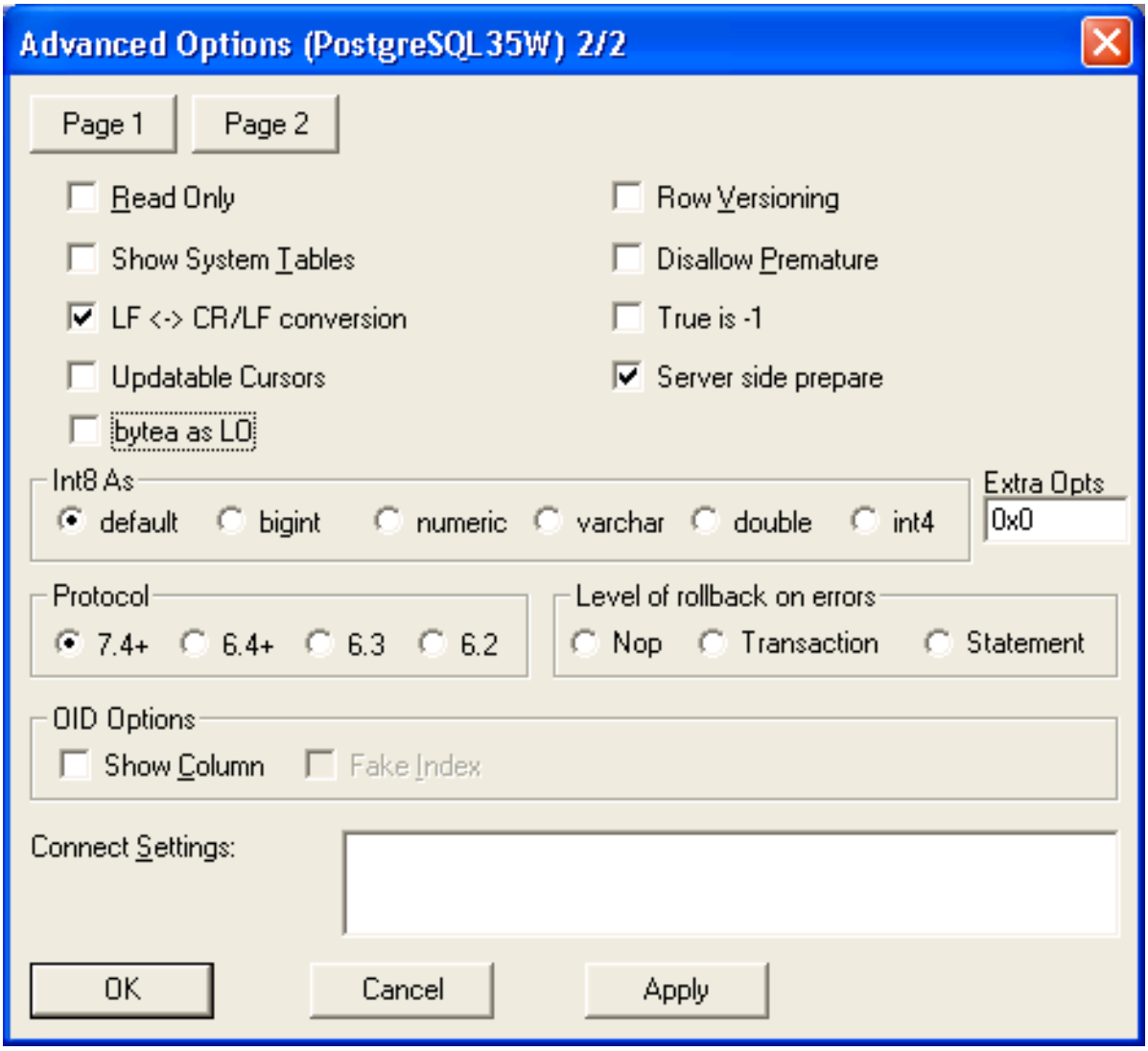


In the *Data Source* Name edit box, type the name you want to assign to this data source. In the *Database* edit box, type the name of the virtual database you want to access through this data source. In the *Server* edit box, type the host name or IP address of your Data Virtualization runtime. If connecting 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 Data Virtualization System listens for ODBC requests. By default, Data Virtualization listens for ODBC requests on port 35432 In the *User Name* and *Password* edit boxes, supply the user name and password for the Data Virtualization runtime access. Provide any description about the data source in the *Description* field.

8. Click on the *Datasource* button, you will see this below figure. Configure options as shown.



Click on "page2" and make sure the options are selected as shown



9. Click "save" and you can optionally click "test" to validate your connection if the Data Virtualization is running. You have configured a Data Virtualization'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

3.4.2. Other *nix Platform Installations

Before you can access Data Virtualization 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 Data Virtualization recommends unixODBC. If you are working with RedHat Linux or Fedora you can check the graphical "yum" installer to search, find and install unixODBC. Otherwise you can download the unixODBC manager here. To install, simply untar the contents of the file to a temporary location and execute the following commands as super user.

```
./configure
make
make install
```

Check unixODBC website site for more information, if you run into any issues during the installation.

Now, to verify that PostgreSQL driver installed correctly from earlier step, execute the following command

```
odbcinst -q -d
```

-

That should show you all the ODBC drivers installed in your system. Now it is time to create a DSN. Edit "/etc/odbc.ini" file and add the following

```
[<DSN name>]
Driver = /usr/lib/psqlodbc.so
Description = PostgreSQL Data Source
Servername = <Data Virtualization 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 Data Virtualization and issue queries an retrieve results.

3.5. DSN LESS CONNECTION

You can also connect to Data Virtualization VDB using ODBC with out explicitly creating a DSN. However, in these scenarios your application needs, what is called as "DSN less connection string". The below is a sample connection string

For Windows:

```
ODBC;DRIVER={PostgreSQL Unicode};DATABASE=<vdb-name>;SERVER=<host-name>;PORT=
<port>;Uid=<username>;Pwd=<password>;c4=0;c8=1;
```

For *nix:

```
ODBC;DRIVER={PostgreSQL};DATABASE=<vdb-name>;SERVER=<host-name>;PORT=
<port>;Uid=<username>;Pwd=<password>;c4=0;c8=1;
```

See the available Data Virtualization connection options.

3.6. 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 Data Virtualization 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 the vdb file as

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

CHAPTER 4. ODATA COMPATIBILITY

4.1. 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

4.2. DATA VIRTUALIZATION COMPATIBILITY FOR ODATA

Data Virtualization is compatible with OData Version 4.0.

4.3. ODATA VERSION 4.0 COMPATIBILITY

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

4.3.1. 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/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.

4.3.2. Query Basics

Users can submit predicates with along their query to filter the results:

```
http://localhost:8080/odata/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/customers?$format=JSON
```

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

```
http://localhost:8080/odata/customers?$filter=name eq 'bob'&$format=xml
```

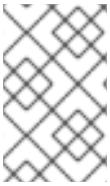
OData allows for querying navigations from one entity to another. A navigation is similar to the foreign key relationships in relational databases.

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/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.

4.3.2.1. How to execute a stored procedure?

Odata allows you to call your exposed stored procedure methods via odata.

```
http://localhost:8080/odata/getcustomersearch(id=120,firstname='micheal')
```

4.3.2.2. 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/customers?$skiptoken=xxx
```

4.3.2.3. "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).

4.3.3. 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
```

4.3.4. Configuration

You can customize the OData interfaces via properties prefixed with "spring.teiid.odata.".

|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

Data Virtualization OData server implements cursoring logic when the result rows exceed the configured batch size. On every request, only *batch-size* number of rows are returned. Each such request is considered an active cursor, with a specified amount of idle time specified by *skip-token-cache-time*. After the cursor is timed out, the cursor will be closed and remaining results will be cleaned up, and will no longer be available for further queries. Since there is no session based tracking of these cursors, if the request for skiptoken comes after the expired time, the original query will be executed again and tries to reposition the cursor to relative absolute position, however the results are not guaranteed to be same as the underlying sources may have been updated with new information meanwhile.

4.3.5. Limitations

The OData4 interface is subject to some feature limitations. You cannot use the following features.

- Search.
- Delta processing.
- Data-aggregation extension of the OData specification.
- $it usage is limited to only primitive collection properties.

4.3.6. 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/

4.3.7. OData Metadata (How Data Virtualization interprets the relational schema into OData's $metadata)

OData defines its schema using Conceptual Schema Definition Language (CSDL). A VDB in an ACTIVE state in Data Virtualization exposes its visible metadata in CSDL format. For example if you want retrieve metadata for your vdb, you need to issue a request like:

```
http://localhost:8080/odata/$metadata
```

Since OData schema model is not a relational schema model, Data Virtualization uses the following semantics to map its relational schema model to OData schema model.

Relational Entity	Mapped OData Entity
Model Name	Schema Namespace, EntityContainer Name
Table/View	EntityType, EntitySet
Table Columns	EntityType's Properties
Primary Key	EntityType's Key Properties
Foreign Key	Navigation Property on EntityType
Procedure	FunctionImport, ActionImport
Procedure's Table Return	ComplexType

Data Virtualization by design does not define any "embedded" ComplexType in the EntityType.

Since OData access is more key based, it is **MANDATORY** that every table Data Virtualization exposes through OData must have a PK or at least one UNIQUE key. A table which does not either of these will be dropped out of the $metadata.

Since all data roles are not consulted in the construction of the OData metadata there are times when tables or procedures will need to be specifically hidden. This can be done in the vdb via a "teiid_odata:visible" extension metadata property on the object.

```
create foreign table HIDDEN (id long primary key, ...) OPTIONS ("teiid_odata:visible" false);
```

With teiid_odata:visible set to false the OData layer will not expose the given object.

Datatype Mapping

Data Virtualization Type	OData Type
STRING	Edm.String
BOOLEAN	Edm.Boolean
BYTE	Edm.SByte
SHORT	Edm.Int16
INTEGER	Edm.Int32
LONG	Edm.Int64
FLOAT	Edm.Single
DOUBLE	Edm.Double
BIG_INTEGER	Edm.Decimal
BIG_DECIMAL	Edm.Decimal
DATE	Edm.Date
TIME	Edm.TimeOfDay
TIMESTAMP	Edm.DateTimeOffset
BLOB	Edm.Stream
CLOB	Edm.Stream
XML	Edm.Stream
VARBINARY	Edm.Binary

Geography and Geometry will be mapped to the corresponding Edm.GeometryXXX and Edm.GeographyXXX types based upon the associated {http://www.teiid.org/translator/spatial/2015}type property. A general mapping to Edm.Geometry or EdmGeography will fail to serialize the values correctly.

Where possible, array types will be mapped to a collection type. However you cannot include multidimensional arrays. Also array/collection values cannot be used as parameters nor in comparisons.

4.3.7.1. Functions And Actions

The mapping of entities and their properties is relatively straight-forward. The mapping of Data Virtualization procedures to OData Functions and Actions is more involved. Virtual procedures, source

procedure, and virtual functions defined by DDL (not a Java class) are all eligible to be mapped. Source functions or virtual functions defined by a Java class are currently not mapped to corresponding OData constructs - please log an issue if you need that functionality. OData does not have an out parameter concept, thus OUT parameters are ignored, and INOUT parameters are treated only as IN. A result set is mapped to a complex type collection result. An array result will be mapped to a simple type collection.

An OData Function will be used if:

- The procedure/function has a return value - either scalar or a result set.
- The procedure/function has no LOB input parameters - currently Clob, Blob, XML, Geometry, Geography, and JSON are considered LOB types.
- The procedure/function is side effect free - this is determined by an explicit value of 0 for the update count. For example: CREATE VIRTUAL PROCEDURE ... OPTIONS (UPDATECOUNT 0) AS BEGIN ...

If any one of those conditions are not met the procedure/function is represented instead by an OData Action. However if there is a result set that has a LOB value, then the procedure is not mapped at all as multiple streaming values cannot be returned.

Note that OData Functions and Actions are called differently. A Function is called by a GET request where the parameter values are included in the URI. An Action is called by a POST where the content provides the parameter values.

Currently only unbounded Functions and Actions are compatible.

You should always consult the $metadata about Functions and Actions to validate how the procedures/functions were mapped.

4.3.8. OpenAPI Metadata

An experimental feature is available to automatically provide a Swagger 2.0 / OpenAPI metadata via [swagger|openapi].json rather than $metadata.

Example OpenAPI 2.0 URLs

```
http://localhost:8080/odata/swagger.json
http://localhost:8080/odata/openapi.json
http://localhost:8080/odata/openapi.json?version=2
```

Example OpenAPI 3.0 URL

```
http://localhost:8080/odata/openapi.json?version=3
```

WARNING

Due to all of the possible query options and expansions this metadata will be significantly larger than the OData EDM representation.

CHAPTER 5. GEOSERVER INTEGRATION

GeoServer is an open source server for geospatial data. It can be integrated with Data Virtualization to serve geospatial data from a variety of sources.

5.1. PREREQUISITES

- Have GeoServer installed. Data Virtualization integration was initially tested with GeoServer version 2.6.x, and is compatible with versions 2.8.x and 2.12.x. See TEIID-5236
- Your Data Virtualization installation should already be setup for pg/ODBC access. This allows the built-in compatibility with GeoServer for PostGIS/PostgreSQL to be used.
- Have a VDB deployed that exposes one or more tables containing an appropriate Geometry column.
    a. The Data Virtualization 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.

5.2. 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 Data Virtualization server host, ODBC port, database (VDB Name with optional version), user, and password, schema (schema/model from the target VDB).
    a. If your VDBs contain target schema or table names with % or _, Data Virtualization 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 Data Virtualization 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 Data Virtualization store.
    a. Note that Data Virtualization is not compatible with the PostGIS function ST_Estimated_Extent and attempts to compute the bounding box from the data, result in log errors.

5.3. ADDITIONAL CONSIDERATIONS

- If you are integrating a PostgreSQL source, you must not re-expose the geometry_columns or geography_columns tables. This is because GeoServer makes unqualified queries that reference geometry_columns and the query should resolve against the Data Virtualization

system table instead.

- Data Virtualization does not by default expose a GT_PK_METADATA, which is optionally used by GeoServer

CHAPTER 6. QGIS INTEGRATION

QGIS is an open source geospatial platform. It can be integrated with Data Virtualization to serve geospatial data from a variety of sources.

6.1. PREREQUISITES

- Have QGIS installed. Data Virtualization integration was last tested with version 2.14.
- Your Data Virtualization installation should already be setup for ODBC access. This allows the built-in compatibility of QGIS for PostGIS/PostgreSQL to be used.
- Have a VDB deployed that exposes one or more tables containing an appropriate Geometry column.
    a. The Data Virtualization 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.

6.2. 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 Data Virtualization server host, ODBC port, database (VDB Name with optional version), user, and password.
    a. If your VDBs contain target schema or table names with % or _, Data Virtualization 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.

6.3. ADDITIONAL CONSIDERATIONS

- If you are integrating a PostgreSQL source, you must not re-expose the postgres system tables including the PostGIS geometry_columns or geography_columns tables. This is because QGIS makes unqualified references to these tables, which may then be ambiguous.
- Operations involving creating or deleting schemas or tables will not work.
- The logs might contain messages related to information_schema.tables - this is to determine if the qgis_editor_widget_styles table exists. Data Virtualization is not compatible with QGIS editor widget styles.

CHAPTER 7. SQLALCHEMY INTEGRATION

SQLAlchemy is an open source SQL toolkit and ORM for Python.

7.1. PREREQUISITES

- Have SQLAlchemy installed installed. Data Virtualization integration was last tested with version 1.1.6.
- Your Data Virtualization installation should already be setup for ODBC access. This allows the built-in compatibility with SQLAlchemy for PostgreSQL to be used.

7.2. USAGE

You should be able to use a SQLAlchemy engine for querying. Reflective import of most table metadata is also provided.

Sample Usage

```
import sqlalchemy
from sqlalchemy import create_engine, Table, MetaData
engine = create_engine("postgresql+psycopg2://user:password@host:35432/vdb")
engine.connect()
#engine is ready for queries
result = connection.execute("select * from some_table")
#reflective table import
meta = MetaData()
test = Table('public.test', meta, autoload=True,
autoload_with=engine,postgresql_ignore_search_path=True)
```

7.3. LIMITATIONS

Only a subset of the PostgreSQL dialect is available. The primary intent is to allow querying through Data Virtualization. If there are additional features that are needed, please log an enhancement request.

Column metadata will not be available for tables that contain the period '.' character. Depending upon your needs, you may need import settings that use simple Data Virtualization names and not source schema qualified names.

7.4. APPLICATION COMPATIBILITY

7.4.1. Superset

Superset is an open source data visualization and dashboard builder. It uses SQLAlchemy to access relational sources.

Once you have followed the above instructions, you may access a Data Virtualization VDB by adding a Database under the Sources menu.

The URL will be of the same form shown in the SQLAlchemy integration:
postgresql+psycopg2://user:password@host:35432/vdb

Basic usage scenarios involving aggregation and all basic types have been tested. If there are additional features that are needed, please log an enhancement request

CHAPTER 8. NODE.JS INTEGRATION

Node.js is an open source event driven runtime that can be integrated with Data Virtualization.

8.1. PREREQUISITES

- Have Node.js installed. The npm pagckage pg is also required. Use "
- Your Data Virtualization installation should already be setup for ODBC access. This allows the optional compatibility with Node.js for PostGIS/PostgreSQL to be used.

8.2. USAGE

For example if you have VDB called "northwind" deployed on your Data Virtualization server, and it has table called "customers" and you are using default configuration such as

user = 'user' password = 'user' host = 127.0.0.1 port = 35432

Simple Access Example

```
const { Client } = require('pg')
const client = new Client({
    user: 'user',
    host: 'localhost',
    database: 'northwind',
    password: 'secretpassword',
    port: 35432,
})
client.connect()

client.query('SELECT CustomerID, ContactName, ContactTitle FROM Customers', (err, res) => {
    console.log(err, res)
    client.end()
})
```

NOTE

you do not have to programmatically specify the connection information in the code as it can be obtained from environment variables and other mechanisms - see https://node-postgres.com

For more information please refer to: https://npmjs.org/package/pg

CHAPTER 9. ADO.NET INTEGRATION

Npgsql is an open source ADO.NET Data Provider for PostgreSQL. It can be integrated with Data Virtualization to provide access from programs written in C#, Visual Basic, F#.

9.1. PREREQUISITES

- Install the Npgsql using the .msi Windows installer. Data Virtualization integration was last tested with version 3.2.6.
- Your Data Virtualization installation should already be setup for pg/ODBC access.
- Have a VDB deployed.

9.2. NPGSQL CONFIGURATION

For information about the available connection parameters, see the Npgsql documentation. Not all configuration parameters have been tested for use with Data Virtualization.

9.3. KNOWN LIMITATIONS

- TEIID-5220 prevents displaying the metadata of tables and views, but does not affect querying. Certain tools, such as PowerBi, may have options to turn of the need to perform metadata introspection.

CHAPTER 10. REAUTHENTICATION

Data Virtualization 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.

CHAPTER 11. EXECUTION PROPERTIES

Execution properties may be set on a per statement basis through the **Data VirtualizationStatement** 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 11.1. Execution Properties

Property Name/String Constant	Description
PROP_TXN_AUTO_WRAP / autoCommitTxn	Same as the connection property.
PROP_PARTIAL_RESULTS_MODE / partialResultsMode	See the Partial Results Mode
RESULT_SET_CACHE_MODE / resultSetCacheMode	Same as the connection property.
SQL_OPTION_SHOWPLAN / SHOWPLAN	Same as the connection property.
NOEXEC / NOEXEC	Same as the connection property.
JDBC4COLUMNNAMEANDLABELSEMANTICS / useJDBC4ColumnNameAndLabelSemantics	Same as the connection property.

CHAPTER 12. 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 Data Virtualization and is handled only in the JDBC client. Since a JDBC clients backs the pg/ODBC transport, it will work there as well.

SET Syntax:

- SET [PAYLOAD] (parameter|SESSION AUTHORIZATION) value
- SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL (READ UNCOMMITTED|READ COMMITTED|REPEATABLE READ|SERIALIZABLE)

Syntax Rules:

- The parameter must be an identifier - it can contain spaces or other special characters only if quoted.
- The value may be either a non-quoted identifier or a quoted string literal value.
- If payload is specified, e.g. "SET PAYLOAD x y", then a session scoped payload properties object will have the corresponding name value pair set. The payload object is not fully session scoped. It will be removed from the session when the XAConnection handle is closed/returned to the pool (assumes the use of Data VirtualizationDataSource). The session scoped payload is superseded by the usage of Data VirtualizationStatement.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'");
```

CHAPTER 13. SHOW STATEMENT

The SHOW statement can be used to see a varitey of information. The SHOW statement is not yet a language feature of Data Virtualization 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.

CHAPTER 14. TRANSACTIONS

Data Virtualization provides three types of transactions from a client perspective:

1. Global
2. Local
3. Request Level

All are implemented by Data Virtualization logically as XA transactions. See the JTA specification for more on XA Transactions.

14.1. LOCAL TRANSACTIONS

A Local transaction from a client perspective affects only a single resource, but can coordinate multiple statements.

14.1.1. 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.

14.1.1.1. Turning Off JDBC Local Transaction Controls

In some cases, tools or frameworks above Data Virtualization 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 Data Virtualization 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.

14.1.2. 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.

14.2. 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 Data Virtualization 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 Data Virtualization 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 Data Virtualization 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

14.2.1. 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 Data Virtualization server. Be sure that the sources that you target support XA or that compensating actions are taken in the event of a failure.

14.3. 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 **Data VirtualizationDataSource** (or Data Virtualization 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 Data Virtualization 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 Data Virtualization XAConnections may participate in the same transaction. The Data Virtualization JDBC XAResource "isSameRM" method returns "true" only if connections are made to the same server instance in a cluster. If the Data Virtualization 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.

14.4. RESTRICTIONS

14.4.1. 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.

14.4.2. Enterprise Information System (EIS) compatibility

The underlying data source that represents the EIS system and the EIS system itself must support XA transactions if they want to participate in distributed XA transaction through Data Virtualization. If source system does not support the XA, then it can not fully participate in the distributed transaction. However, the source is still eligible to participate in data integration without the XA support.

The participation in the XA transaction is automatically determined based on the source XA capability. It is user's responsibility to make sure that they configure a XA resource when they require them to participate in distributed transaction.