Progress DataDirect for JDBC for Denodo

User's Guide

Release 6.0.0

Copyright

Table of Contents

1

Welcome to the Progress DataDirect for JDBC for Denodo Driver

The Progress® DataDirect® for JDBC™ for Denodo™ driver (Denodo driver) supports the JDBC API for SQL read-write access to the Denodo Data Virtualization platform.

The documentation for the driver also includes the *Progress DataDirect for JDBC Drivers Reference*. The reference provides general reference information for all DataDirect drivers for JDBC, including content on troubleshooting, supported SQL escapes, and DataDirect tools.

For the complete documentation set, visit the Progress DataDirect Connectors Documentation Hub: https://docs.progress.com/category/datadirect-denodo.

For details, see the following topics:

What's new in this release?

Support and certification

Visit the following web pages for the latest support and certification information.

- Release Notes: https://www.progress.com/datadirect-connectors/whats-new#jdbc
- DataDirect Product Compatibility Guide: https://docs.progress.com/bundle/datadirect-product-compatibility/resource/datadirect-product-compatibility.pdf

Highlights of 6.0.0 Release

- The driver supports SQL read-write access to the Denodo Data Virtualization platform.
- The driver supports JDBC core functions. For details, refer to "JDBC support" in the *Progress DataDirect for JDBC Drivers Reference*.
- The driver supports user ID and password authentication. See Authentication on page 41 for more information.
- The driver supports executing create, read, update, and delete (CRUD) operations on wrapper tables.
- The driver supports TLS/SSL encryption. See Data Encryption on page 42 for more information.
- The driver provides failover support. See Failover on page 45 for more information.
- The driver provides proxy support. See Proxy server example on page 14 for more information.

Requirements

The driver is compatible with JDBC 2.0, 3.0, and 4.0.

The driver requires a Java Virtual Machine (JVM) that is Java SE 8 or higher, including Oracle JDK, OpenJDK, and IBM SDK (Java) distributions.

Installing and setting up the driver

This section provides you with an overview of the steps required to install and set-up the driver. After completing this procedure, you will be able to begin accessing data with your application.

To begin accessing data with the driver:

1. Install the driver:
   a) After downloading the product, unzip the installer files to a temporary directory.
   b) From the installer directory, run the appropriate installer file to start the installer.
      - **Windows**: `PROGRESS_DATADIRECT_JDBC_INSTALL.exe`

- **Non-Windows**: `PROGRESS_DATADIRECT_JDBC_INSTALL.jar`

c) Follow the prompts to complete installation.

The installer program supports multiple installation methods, including command-line and silent installations. For detailed instructions, refer to the *Progress DataDirect for JDBC Drivers Installation Guide*.

2. Set your system CLASSPATH to include the driver `.jar` file. The CLASSPATH is the search string your Java Virtual Machine (JVM) uses to locate JDBC drivers on your computer. The following examples demonstrate setting the CLASSPATH from a command line using the default installation directory.

   - **Windows Example**

     ```
     CLASSPATH=.;C:\Program Files\Progress\DataDirect\JDBC\lib\60\denodo.jar
     ```

   - **UNIX/LINUX Example**

     ```
     CLASSPATH=.:/opt/Progress/DataDirect/JDBC/lib/60/denodo.jar
     ```

3. Configure your driver using one of the following methods:

   - **Connection URL**: You can begin using the driver immediately by passing a connection URL with your application or tool. The following example shows how to connect using the user ID/password authentication.

     ```
     jdbc:datadirect:denodo://myserver:9996;
     DatabaseName=mydb;User=jsmith;Password=secret;
     ```

     Note: The User and Password properties are not required to be stored in the connection string. They can also be passed separately by the application.

     Note: See Authentication on page 41 for details.

   - **Data sources**: The driver also supports connecting using JDBC data sources. A JDBC data source is a Java object, specifically a DataSource object, that defines connection information required for a JDBC driver to connect to the database. See Connecting using data sources for more information.

   Note: For most connections, specifying the minimum required connection properties is sufficient to begin accessing data; however, you can provide values for optional properties to use additional supported features and improve performance.

4. Set the values for any optional properties that you want to configure. For additional information on optional features and functionality, see the following resources:

   - Connection URL examples provides connection string examples that can be used to configure common functionality and features. You can modify and combine these examples to create a string that best suites your environment.
   - Connection property descriptions provides a complete list of supported properties by functionality.

- Performance considerations describes connection properties that affect performance, along with recommended settings.

5. Connect to your service and begin accessing data with your applications, BI tools, database tools, and more. To help you get started, the following resources guide you through accessing data with some common tools:

   - Tableau: Tableau is a business intelligence software program that allows you to easily create reports and visualized representations of your data.
   - DbVisualizer: DB Visualizer is a database tool that allows you to connect and execute SQL statements against your data.
   - Interactive SQL for JDBC (JDBCISQL): JDBCISQL is a command line interface that allows you to connect to a data source, execute SQL statements and retrieve results for display on a terminal.
   - DataDirect Test: DataDirect Test allows you to test connect, execute SQL statements, and practice using the JDBC API right out of the box.

This completes the deployment of the driver.

Driver and DataSource classes

The following are the `Driver` and `DataSource` classes used by the driver:

Driver class:

com.ddtek.jdbc.denodo.DenodoDriver

DataSource class:

com.ddtek.jdbcx.denodo.DenodoDataSource

Connection URL examples

After setting the CLASSPATH, the connection information needs to be passed in the form of a connection URL. This section provides examples of connection strings configured to use common features and functionality. You can modify and/or combine these examples to create a connection string for your environment.

Note:

- Connection property names are case-insensitive. For example, `Password` is the same as `password`.
- For connection properties that support string values, use the following escape sequence to specify values containing leading or trailing spaces and curly brackets: `{value}`. For example: `User={hello }` or `Password={{hello}}`.

User ID/password authentication

This string includes the properties used to connect with the user ID/password authentication.

```
jdbc:datadirect:denodo://servername:port;DatabaseName=database;
User=userID;Password=password;[property=value[;...]];
```

where:

servername

is the IP address or name of the server to which you are connecting.

port

is the number of the TCP/IP port.

database

is the name of the database you want to connect to.

userID

specifies the user ID that is used to connect to the Denodo database.

password

specifies a password that is used to connect to your Denodo database.

property=value

specifies connection property settings. Multiple properties are separated by a semi-colon.

Note: The User and Password properties are not required to be stored in the connection string. They can also be passed separately by the application.

The following example connection string includes the properties required for connecting with the user ID/password authentication.

```
Connection conn = DriverManager.getConnection
  ("jdbc:datadirect:denodo://myserver:9996;DatabaseName=mydb;
    User=jsmith;Password=secret;");
```

See also

Connection property descriptions on page 57
Authentication on page 41

Proxy server example

This string includes the properties you may need to connect through a proxy server with user ID/password authentication.

```
jdbc:datadirect:denodo://servername:port;DatabaseName=database;ProxyHost=proxy_host;
ProxyPassword=proxy_password;ProxyPort=proxy_port;ProxyUser=proxy_user;
User=user_name;Password=password;[property=value[;...]];
```

where:

servername

specifies either the IP address in IPv4 or IPv6 format, or the server name (if your network supports named servers) of the primary database server.

port

specifies the number of the TCP/IP port.

database

specifies the name of the database you want to connect to.

proxy_host

specifies the proxy server to use for the first connection.

proxy_password

specifies the password needed to connect to a proxy server for the first connection.

proxy_port

specifies the port number where the proxy server is listening for requests for the first connection. The default is `0`.

proxy_user

specifies the user name needed to connect to a proxy server for the first connection.

user_name

specifies the user name that is used to connect to the server. For example, `jsmith`.

password

specifies the password used to connect to the server.

property=value

specifies connection property settings. Multiple properties are separated by a semi-colon.

Note: The User and Password properties are not required to be stored in the connection string. They can also be passed separately by the application.

The following example connection strings include the properties required for using a proxy server with user ID/password authentication.

```
Connection conn = DriverManager.getConnection
  ("jdbc:datadirect:denodo://myserver:9996;DatabaseName=mydb;
    ProxyHost=pserver;ProxyPassword=proxypwd;ProxyPort=123;ProxyUser=johndoe;
    User=jsmith;Password=secret;");
```

See also

Connection property descriptions on page 57

Data types

The following table lists the data types supported by the Denodo driver and describes how they are mapped to JDBC data types.

Table 1: Denodo Data Types

Denodo Data Type	JDBC Data Type
Blob	BLOB or BINARY or VARBINARY or LONGVARBINARY
Boolean	BOOLEAN or BIT
Decimal	DECIMAL or NUMERIC
Double	DOUBLE
Float	FLOAT or REAL
Int	INTEGER or SMALLINT or TINYINT
Localdate	DATE
Long	BIGINT
Text	VARCHAR or CHAR or LONGVARCHAR
Time	TIME
Timestamp	TIMESTAMP
Timestamptz	TIMESTAMP_WITH_TIMEZONE
XML	SQLXML

getTypeInfo()

The following table provides getTypeInfo() results for Denodo databases supported by the driver.

Table 2: getTypeInfo() for Denodo

TYPE_NAME = blob AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 2004 (BLOB) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = Blob MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 2147483647 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = true
TYPE_NAME = boolean AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 16 (BOOLEAN) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = Boolean MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 1 SEARCHABLE = 2 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = true
TYPE_NAME = decimal AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = precision,scale DATA_TYPE = 3 (DECIMAL) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = Decimal MAXIMUM_SCALE = 20	MINIMUM_SCALE = 0 Decimal NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 38 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false

TYPE_NAME = double AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 8 (DOUBLE) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = Double MAXIMUM_SCALE = 0	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 15 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false
TYPE_NAME = float AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 6 (FLOAT) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = Float MAXIMUM_SCALE = 0	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 7 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false
TYPE_NAME = int AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 4 (INTEGER) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = Int MAXIMUM_SCALE = 0	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 10 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false

TYPE_NAME = localdate AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 91 (DATE) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = Localdate MAXIMUM_SCALE = 0	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 10 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = true
TYPE_NAME = long AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = -5 (BIGINT) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = Long MAXIMUM_SCALE = 0	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 19 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false
TYPE_NAME = text AUTO_INCREMENT = NULL CASE_SENSITIVE = true CREATE_PARAMS = NULL DATA_TYPE = 12 (VARCHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = Text MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 65536 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = true

TYPE_NAME = time AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 92 (TIME) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = Time MAXIMUM_SCALE = 6	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 26 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = true
TYPE_NAME = timestamp AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 93 (TIMESTAMP) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = Timestamp MAXIMUM_SCALE = 6	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 26 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = true
TYPE_NAME = timestamptz AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 2014 (TIMESTAMP_WITH_TIMEZONE) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = Timestamptz MAXIMUM_SCALE = 6	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 32 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = true

TYPE_NAME = date AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 91 (DATE) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = DATE MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 10 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = XML AUTO_INCREMENT = NULL CASE_SENSITIVE = true CREATE_PARAMS = NULL DATA_TYPE = 2009 (SQLXML) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = XML MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 65536 SEARCHABLE = 0 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = true

SQL escape sequences

The driver supports the following SQL escape sequences.

- Date, Time, and Timestamp Escape Sequences
- Scalar Functions
- Outer Join Escape Sequences
- LIKE Escape Character Sequence for Wildcards

Refer to "SQL escape sequences" in the *Progress DataDirect for JDBC Drivers Reference* for information about SQL escape sequences.

Supported scalar functions

You can use scalar functions in SQL statements with the following syntax:

{fn *scalar-function*}

where:

scalar-function

is a scalar function supported by the driver, as listed in the following table.

Example:

```
SELECT id, name FROM emp WHERE name LIKE {fn UCASE('Smith')}
```

Table 3: Supported Scalar Functions

String Functions	Numeric Functions	Timedate Functions	System Functions
ASCII	ABS	CURRENT_DATE	IFNULL
CHAR	ACOS	DAYOFMONTH	GETSESSION
CONCAT	ASIN	DAYOFWEEK	NULLIF
LCASE	ATAN	DAYOFYEAR	ROWNUM
LOCATE	ATAN2	EXTRACT	
LTRIM	COS	MONTH	
REPEAT	COT	MONTHNAME	
REPLACE	DEGREES	NOW	
RIGHT	EXP	QUARTER	
RTRIM	FLOOR	WEEK	
SUBSTRING	LOG	YEAR	
UCASE	LOG10	ADDDAY	
CHAR_LENGTH	MOD	ADDHOUR	
INSTR	PI	ADDMINUTE	
LEN	POWER	ADDMONTH	
LOWER	RADIANS	ADDSECOND	
MAX	ROUND	ADDWEEK	
MIN	SIGN	ADDYEAR	
POSITION	SIN	FIRSTDAYOFMONTH	
REGEXP	SQRT	FIRSTDAYOFWEEK	
REMOVEACCENTS	TAN	FORMATDATE	
REPLACEMAP	TRUNCATE	GETDAY	
SPLIT	CEIL	GETDAYOFWEEK	
TEXTCONSTANT	DIV	GETDAYOFYEAR	
TRIM	LN	GETDAYSBETWEEN	
UPPER	MAX	GETHOUR	

String Functions	Numeric Functions	Timedate Functions	System Functions
	MIN MULT SUBTRACT SUM	GETMILLISECOND GETMINUTE GETMONTH GETMONTHSBETWEEN GETQUARTER GETSECOND GETTIMEINMILLIS GETWEEK GETYEAR LASTDAYOFMONTH LASTDAYOFWEEK MAX MIN NEXTWEEKDAY PREVIOUSWEEKDAY SUBTRACT TO_DATE TRUNC	

DataDirect tools

Progress DataDirect for JDBC drivers install the set of tools described in this section. For detailed instructions on using these tools, refer to the corresponding topics in the *Progress DataDirect for JDBC Drivers Reference*.

- DataDirect Test allows you to test your JDBC driver and learn the JDBC API.
- DataDirect Connection Pool Manager allows you to pool connections when accessing databases. When your applications use connection pooling, connections are reused rather than created each time a connection is requested. Because establishing a connection is among the most costly operations an application may perform, using Connection Pool Manager to implement connection pooling can significantly improve performance.
- Statement Pool Monitor loads statements into and remove statements from the statement pool as well as generate information to help you troubleshoot statement pooling performance.
- DataDirect Spy logs detailed information about calls your driver makes that can be used for troubleshooting.

Troubleshooting

The *Progress DataDirect for JDBC Drivers Reference* provides information on troubleshooting problems should they occur. Refer to the "Troubleshooting" section in the *Reference* for details.

Additional information

In addition to the content provided in this guide, the documentation set also contains detailed conceptual and reference information that applies to all the drivers. For more information in these topics, refer the *Progress DataDirect for JDBC Drivers Reference* or use the links below to view some common topics:

- "JDBC support" describes support for JDBC interfaces and methods for the Progress DataDirect for JDBC drivers.
- "JDBC extensions" describes the JDBC extensions provided by the com.ddtek.jdbc.extensions package.
- "SQL escape sequences for JDBC" provides an overview of SQL escape sequences for JDBC. In addition, it documents the scalar functions that you use in SQL statements.
- "Security best practices for JDBC applications" describes the security best practices you should employ when developing and deploying your application with the driver.

Contacting Technical Support

Progress DataDirect offers a variety of options to meet your support needs. Please visit our Web site for more details and for contact information:

https://www.progress.com/support

The Progress DataDirect Web site provides the latest support information through our global service network. The SupportLink program provides access to support contact details, tools, patches, and valuable information, including a list of FAQs for each product. In addition, you can search our Knowledgebase for technical bulletins and other information.

When you contact us for assistance, please provide the following information:

- Your number or the serial number that corresponds to the product for which you are seeking support, or a case number if you have been provided one for your issue. If you do not have a SupportLink contract, the SupportLink representative assisting you will connect you with our Sales team.
- Your name, phone number, email address, and organization. For a first-time call, you may be asked for full information, including location.
- The Progress DataDirect product and the version that you are using.
- The type and version of the operating system where you have installed your product.
- Any database, database version, third-party software, or other environment information required to understand the problem.
- A brief description of the problem, including, but not limited to, any error messages you have received, what steps you followed prior to the initial occurrence of the problem, any trace logs capturing the issue, and so

on. Depending on the complexity of the problem, you may be asked to submit an example or reproducible application so that the issue can be re-created.

- A description of what you have attempted to resolve the issue. If you have researched your issue on Web search engines, our Knowledgebase, or have tested additional configurations, applications, or other vendor products, you will want to carefully note everything you have already attempted.
- A simple assessment of how the severity of the issue is impacting your organization.

2

Tutorials

The following sections guide you through using the driver to access your data with some common third-party applications. For information on installing your driver and setting the CLASSPATH, see "Installing and setting-up the driver."

For details, see the following topics:

- Tableau
- DbVisualizer
- Interactive SQL for JDBC (JDBCISQL)

Tableau

After you have installed your driver and defined it on the CLASSPATH, you can use the driver to access your data with Tableau. Tableau is a business intelligence software program that allows you to easily create reports and visualized representations of your data. By using the driver with Tableau, you can improve performance when retrieving data while leveraging the driver's relational mapping tools.

To use the driver to access data with Tableau:

1. Navigate to the `\lib\xx` subdirectory of the Progress DataDirect installation directory; then, locate the `jar` file for your driver:

   ```
   denodo.jar
   ```

2. Copy the `.jar` file for your driver into the following directory:

Windows: `C:\Program Files\Tableau\Drivers`

Linux: `/opt/tableau/tableau_driver/jdbc`

3. Open Tableau. From the **Connect** menu, select **Other Databases (JDBC)**.
4. In the **Other Databases (JDBC)** dialog, provide values for the following fields; then, click **Sign In**.
   - **URL**: Copy and paste your connection URL into this field. The following example shows how to connect using the user ID/password authentication.

     ```
     jdbc:datadirect:denodo://myserver:9996;DatabaseName=mydb
     ```

     Note: See Authentication on page 41 for details.

   - **Dialect**: Select **SQL92** (the default) from the drop-down box.
   - **Username**: If required by the authentication method being used, enter the user name. Alternatively, this value can be specified with the `User` property in the connection string.
   - **Password**: If required by the authentication method being used, enter the password. Alternatively, this value can be specified with the `Password` property in the connection string.
5. The **Data Source** window appears. In the **Schema** field, select the schema for the service you want to use.
6. In the **Table** field, the tables stored in the selected schema are now exposed and available for selection.

You have successfully accessed your data and are now ready to create reports with Tableau. For detailed information, refer to the Tableau product documentation at: https://www.tableau.com/support/help.

DbVisualizer

After you have installed your driver and defined it on the CLASSPATH, you can use the driver to access your data with the third-party DbVisualizer tool. The following topics guide you through using DbVisualizer to add your driver, connect, and execute SQL statements.

Adding a driver

To add a driver with DbVisualizer:

1. Open DbVisualizer.
2. From the menu, select **Tools**>**Driver Manager**. The Driver Manager window opens.
3. From the Driver Manager menu, select **Driver**>**Create Driver**.
4. Click the  button to navigate to the location of the driver jar file; then, click **OK**. The following are the default locations for the driver:

   Windows

   ```
   C:\Program Files\Progress\DataDirect\JDBC\lib\60\denodo.jar
   ```

Linux

```
/opt/Progress/DataDirect/JDBC/lib/60/denodo.jar
```

5. Provide values for the following fields; then, close the Driver Manager window.
   - **Name**: Type an alias for your driver. For example:

     ```
     Denodo
     ```

   - **URL Format**: Optionally, specify the format of the connection string for your driver. For example:

     ```
     jdbc:datadirect:denodo://myserver:9996;DatabaseName=mydb
     ```

   - **Driver Class**: From the drop down menu, select the driver class for your driver. For example:

     ```
     com.ddtek.jdbc.denodo.DenodoDriver
     ```

You can now use your driver with DbVisualizer. Proceed to "Connecting and executing SQL statements" for information on connecting and executing SQL statements.

Connecting and executing SQL statements

To use the driver to access data with DbVisualizer:

1. Open DbVisualizer.
2. From the menu, select **Database**>**New Connection**. When prompted to use the Connection Wizard, click **OK**.
3. Provide the following information when prompted; then, click **Next** to proceed:
   - **Connection alias**: Type the name to be used when referring to this connection.
   - **Driver**: Select the alias that you provided for your driver from the drop-down menu.
4. Provide values for the following fields; then, click **Finish**.
   - **Database URL**: Copy and paste your connection URL into this field. The following example shows how to connect using the user ID/password authentication.

     User ID/password authentication

     ```
     jdbc:datadirect:denodo://myserver:9996;
     DatabaseName=mydb;User=test;Password=secret;
     ```

     Note: See Authentication on page 41 for details.

5. To execute SQL statements, select **SQL Commander**>**New SQL Commander**. A SQL Commander tab opens.
6. Select values for the following fields:
   - **Database Connection**: Select connection alias you provided for the connection from the drop-down menu.

- **Schema**: Select the schema you want to execute queries against from the drop-down menu.

7. In the SQL Commander tab, enter SQL commands you want to execute; then select **SQL Commander**>**Execute**. For example:

   To select all of the rows from the BLOGS table:

   ```
   SELECT * BLOGS
   ```

You have successfully accessed your data with DbVisualizer.

Interactive SQL for JDBC (JDBCISQL)

After you have installed your driver and defined it on the CLASSPATH, you can use the driver to access your data with Interactive SQL for JDBC (JDBCISQL). JDBCISQL supports a command line interface that allows you to connect to a data source, execute SQL statements and retrieve results for display on a terminal.

To execute commands with JDBCISQL:

1. Start the ISQL tool. Do one of the following:

   - On Windows, double-click the`jdbcisql.bat` file in the *`install_dir`*`\jdbcisql` folder. Or, from a command prompt, navigate to the *`install_dir`*`\jdbcisql` directory and run the `jdbcisql.bat` file.
   - On Linux and UNIX, change to the *`install_dir`*`\isql` directory and run `jdbcisql.sh`.

   The Interactive SQL prompt appears.

2. Type the driver name class; then, press **Enter**:

   ```
   com.ddtek.jdbc.denodo.DenodoDriver
   ```

3. Type `connect` followed by the connection URL for the driver; then, press **Enter**. For example:

   ```
   connect jdbc:datadirect:denodo://myserver:9996;
   DatabaseName=mydb;User=jsmith;Password=secret;
   ```

   If successful, the tool will return the time required to connect.

4. At the `ISQL>` prompt, issue a SQL command to query or modify the data source; then, press **Enter**. For example:

   ```
   SELECT * FROM ACCOUNT;
   ```

   Note: SQL commands must be terminated by a semi-colon.

   Note: In addition to SQL commands, JDBCISQL supports a set of proprietary commands. Type `Help` at the prompt for a list of supported commands and syntax.

   The results of the command are displayed in the terminal.

5. After you are finished executing queries and commands, you can disconnect from the data source by typing the following; then, pressing **Enter**:

   ```
   DISCONNECT;
   ```

6. Press any key to end the session.

3

Configuring and connecting

This section provides information on how to connect to your data store using either the JDBC Driver Manager or DataDirect JDBC data sources, as well as information on how to implement and use functionality supported by the driver.

After the driver has been installed and defined on your classpath, you can connect from your application to your data in either of the following ways.

- Using the JDBC `DriverManager` by specifying the connection URL in the `DriverManager.getConnection()` method.
- Creating a JDBC data source that can be accessed through the Java Naming Directory Interface (JNDI).

For details, see the following topics:

Setting the classpath

The driver must be defined on your CLASSPATH before you can connect. The CLASSPATH is the search string your Java Virtual Machine (JVM) uses to locate JDBC drivers on your computer. If the driver is not defined on your CLASSPATH, you will receive a `class not found` exception when trying to load the driver. Set your system CLASSPATH to include the driver jar file as shown, where *`install_dir`* is the path to your product installation directory.

`install_dir``/lib/60/denodo.jar`

Windows Example

```
CLASSPATH=.;C:\Program Files\Progress\DataDirect\JDBC\lib\60\denodo.jar
```

UNIX Example

```
CLASSPATH=.:/opt/Progress/DataDirect/JDBC/lib/60/denodo.jar
```

Connecting using the JDBC Driver Manager

One way to connect to a service is through the JDBC DriverManager using the `DriverManager.getConnection()` method. As the following example shows, this method specifies a string containing a connection URL.

User ID/password authentication

```
Connection conn = DriverManager.getConnection
  ("jdbc:datadirect:denodo://myserver:9996;
    DatabaseName=mydb;User=jsmith;Password=secret;");
```

Note: The User and Password properties are not required to be stored in the connection string. They can also be passed separately by the application.

Passing the connection URL

After setting the CLASSPATH, the required connection information needs to be passed in the form of a connection URL. The following example includes the properties required for connecting with user ID/password authentication.

Connection URL Syntax

The connection URL takes the following form:

```
jdbc:datadirect:denodo://servername:port;DatabaseName=database;
User=userID;Password=password;[property=value[;...]];
```

where:

servername

is the IP address or name of the server to which you are connecting.

port

is the number of the TCP/IP port.

database

is the name of the database to which you want to connect.

userID

specifies the user ID that is used to connect to the Denodo database.

password

specifies a password that is used to connect to your Denodo database.

property=value

specifies connection property settings. Multiple properties are separated by a semi-colon.

The following example connection string includes the properties required for connecting with the user ID/password authentication.

```
Connection conn = DriverManager.getConnection
  ("jdbc:datadirect:denodo://myserver:9996;
    DatabaseName=mydb;User=jsmith;Password=secret;");
```

Note: The User and Password properties are not required to be stored in the connection string. They can also be passed separately by the application.

Note: The driver connects to the 9996 port and any customized port that is configured on the PostgreSQL wire protocol.

See also

Connection property descriptions on page 57
Connection URL examples on page 12
Authentication on page 41

Testing the connection

You can use DataDirect Test™ to verify your connection. The screen shots in this section were taken on a Windows system.

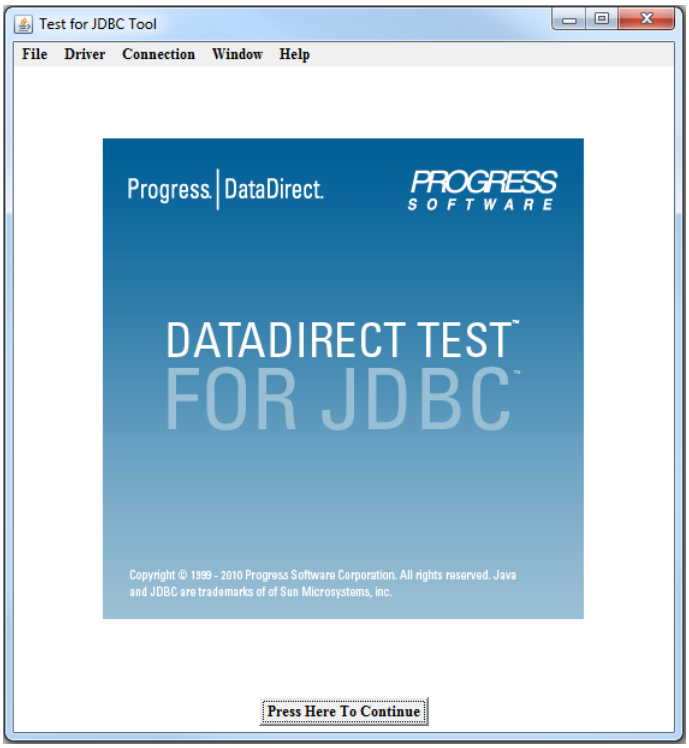
To test the connection from the driver to your data source, follow these steps:

1. Navigate to the installation directory. The default location is:
   - Windows systems: `Program Files\Progress\DataDirect\JDBC_60\testforjdbc`
   - UNIX and Linux systems: `/opt/Progress/DataDirect/JDBC_60/testforjdbc`

Note: For UNIX/Linux, if you do not have access to `/opt`, your home directory will be used in its place.

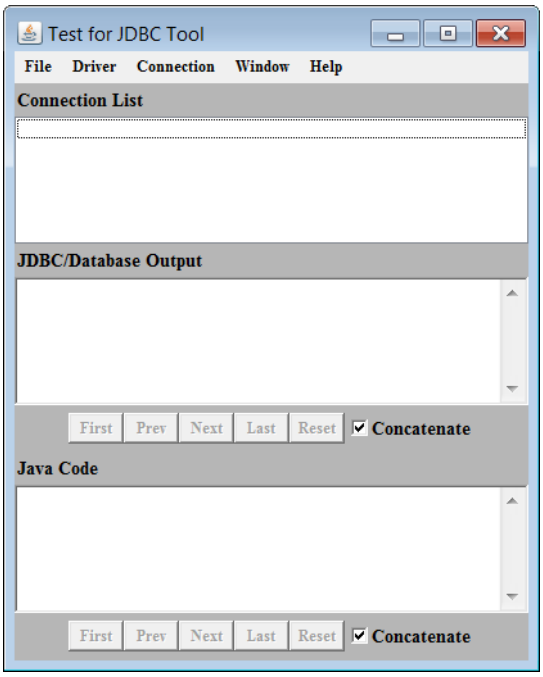
2. From the `testforjdbc` folder, run the platform-specific tool:
   - `testforjdbc.bat` (on Windows systems)
   - `testforjdbc.sh` (on UNIX and Linux systems)

   The **Test for JDBC Tool** window appears:



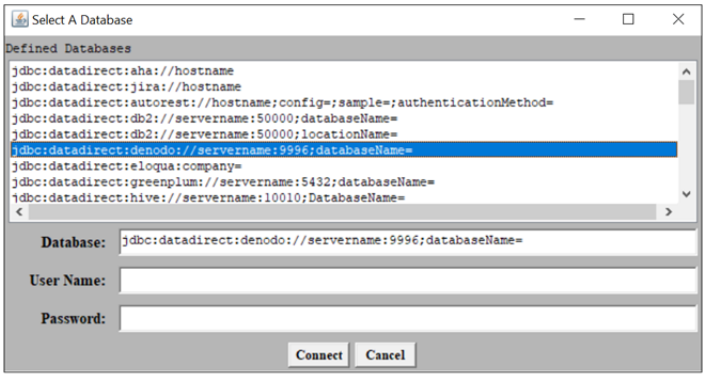
3. Click **Press Here to Continue**.

   The main dialog appears:



4. From the menu bar, select **Connection > Connect to DB**.

   The **Select A Database** dialog appears:



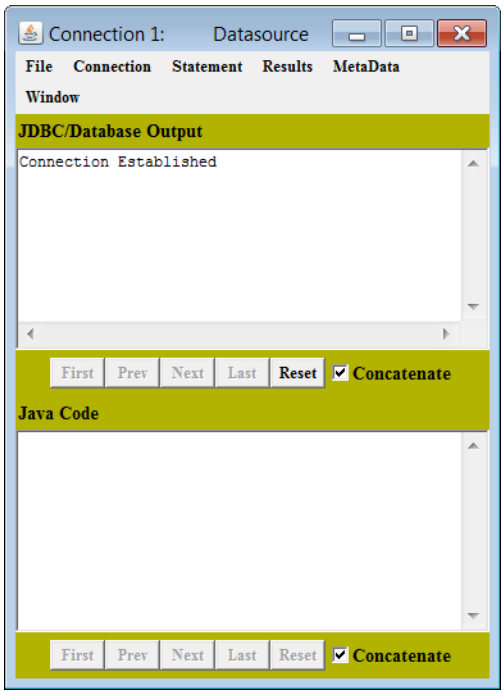
5. Select the appropriate database template from the **Defined Databases** field.
6. In the **Database** field, specify all required connection properties.

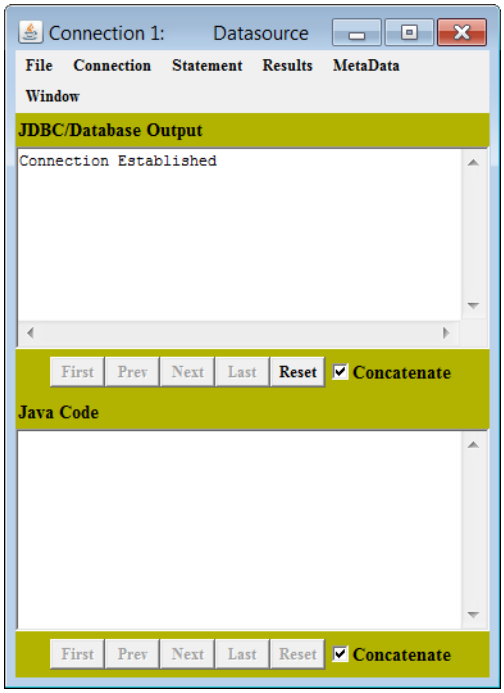
   For example:

   ```
   jdbc:datadirect:denodo://myserver:9996;DatabaseName=mydb
   ```

7. Click **Connect**.

If the connection information is entered correctly, the **JDBC/Database Output** window reports that a connection has been established. (If a connection is not established, the window reports an error.)





Refer to "DataDirect Test" in the *Progress DataDirect for JDBC Drivers Reference* for more information about using DataDirect Test.

Connecting using data sources

A *JDBC data source* is a Java object, specifically a `DataSource` object, that defines connection information required for a JDBC driver to connect to the database. Each JDBC driver vendor provides their own data source implementation for this purpose. A Progress DataDirect data source is Progress DataDirect's implementation of a `DataSource` object that provides the connection information needed for the driver to connect to a database.

Because data sources work with the Java Naming Directory Interface (JNDI) naming service, data sources can be created and managed separately from the applications that use them. Because the connection information is defined outside of the application, the effort to reconfigure your infrastructure when a change is made is minimized. For example, if the database is moved to another database server, the administrator need only change the relevant properties of the `DataSource` object. The applications using the database do not need to change because they only refer to the name of the data source.

How data sources are implemented

Data sources are implemented through a data source class. A data source class implements the following interfaces.

- `javax.sql.DataSource`
- `javax.sql.ConnectionPoolDataSource` (allows applications to use connection pooling)

Refer to "Connection Pool Manager" in the *Progress DataDirect for JDBC Drivers Reference* for more information.

See also

Driver and DataSource classes on page 12

Creating data sources

The following example files provide details on creating and using Progress DataDirect data sources with the Java Naming Directory Interface (JNDI), where *`install_dir`* is the product installation directory.

- *`install_dir`*`/Examples/JNDI/JNDI_LDAP_Example.java` can be used to create a JDBC data source and save it in your LDAP directory using the JNDI Provider for LDAP.
- *`install_dir`*`/Examples/JNDI/JNDI_FILESYSTEM_Example.java` can be used to create a JDBC data source and save it in your local file system using the File System JNDI Provider.

See "Example data source" for an example data source definition for the example files.

To connect using a JNDI data source, the driver needs to access a JNDI data store to persist the data source information. For a JNDI file system implementation, you must download the File System Service Provider from the Oracle Technology Network Java SE Support downloads page, unzip the files to an appropriate location, and add the `fscontext.jar` and `providerutil.jar` files to your CLASSPATH. These steps are not required for LDAP implementations because the LDAP Service Provider is included with supported versions of Java SE.

Example data source

To configure a data source using the example files, you will need to create a data source definition. The content required to create a data source definition is divided into three sections.

First, you will need to import the data source class. For example:

```
import com.ddtek.jdbcx.denodo.DenodoDataSource;
```

Next, you will need to set the values and define the data source. For example, the following definition contains the minimum properties required to establish connection:

Note:

- Setting the password using a data source is generally not recommended. The data source persists all properties, including the Password property, in clear text.
- In a JDBC data source, string values must be enclosed in double quotation marks, for example, `setUser("abc@defcorp.com")`.

```
DenodoDataSource mds = new DenodoDataSource();
mds.setDescription("My Denodo Data Source");
mds.setServerName("myserver");
mds.setPortNumber("9996");
mds.setDatabaseName("mydb");
```

Finally, you will need to configure the example application to print out the data source attributes. Note that this code is specific to the driver and should only be used in the example application. For example, you would add the following section for the minimum properties required to establish a connection:

```
if (ds instance of DenodoDataSource)
{
DenodoDataSource jmds = (DenodoDataSource) ds;
System.out.println("description=" + jmds.getDescription());
System.out.println("serverName=" + jmds.getServerName());
System.out.println("portNumber=" + jmds.getPortNumber());
System.out.println("databaseName=" + jmds.getDatabaseName());
System.out.println();
}
```

Calling a data source in an application

Applications can call a Progress DataDirect data source using a logical name to retrieve the `javax.sql.DataSource` object. This object loads the specified driver and can be used to establish a connection to the database.

Once the data source has been registered with JNDI, it can be used by your JDBC application as shown in the following code example.

```
Context ctx = new InitialContext();
DataSource ds = (DataSource)ctx.lookup("EmployeeDB");
Connection con = ds.getConnection("domino", "spark");
```

In this example, the JNDI environment is first initialized. Next, the initial naming context is used to find the logical name of the data source (`EmployeeDB`). The `Context.lookup()` method returns a reference to a Java object, which is narrowed to a `javax.sql.DataSource` object. Then, the `DataSource.getConnection()` method is called to establish a connection.

Testing a data source connection

You can use DataDirect Test™ to establish and test a data source connection. The screen shots in this section were taken on a Windows system.

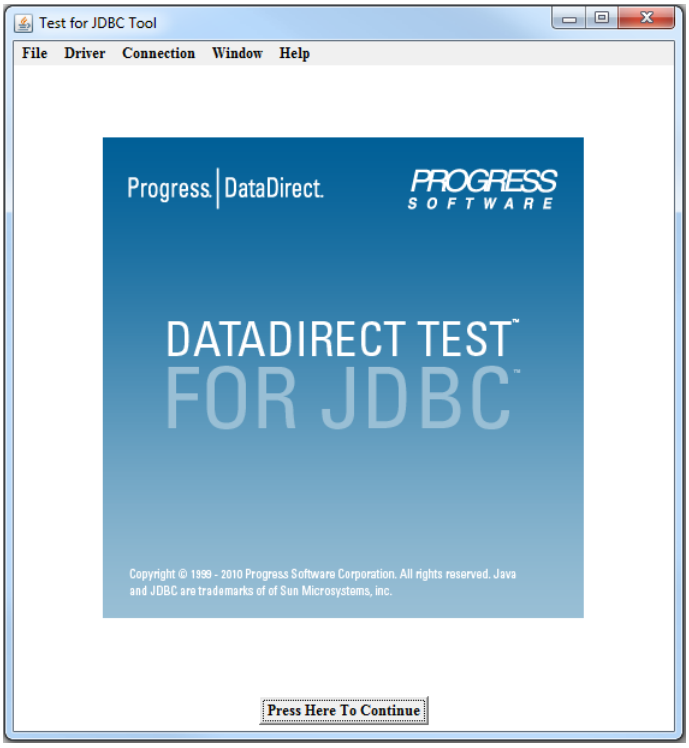
Take the following steps to establish a connection.

1. Navigate to the installation directory. The default location is:
   - Windows systems: `Program Files\Progress\DataDirect\JDBC\testforjdbc`
   - UNIX and Linux systems: `/opt/Progress/DataDirect/JDBC/testforjdbc`

   Note: For UNIX/Linux, if you do not have access to `/opt`, your home directory will be used in its place.

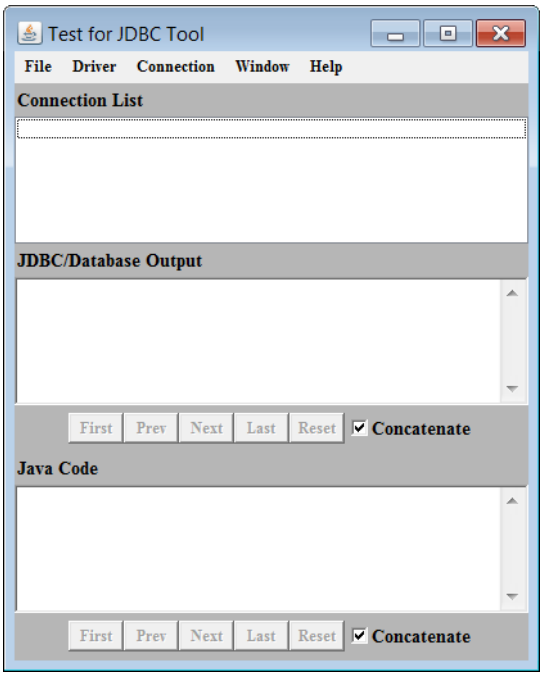
2. From the `testforjdbc` folder, run the platform-specific tool:
   - `testforjdbc.bat` (on Windows systems)
   - `testforjdbc.sh` (on UNIX and Linux systems)

   The **Test for JDBC Tool** window appears:



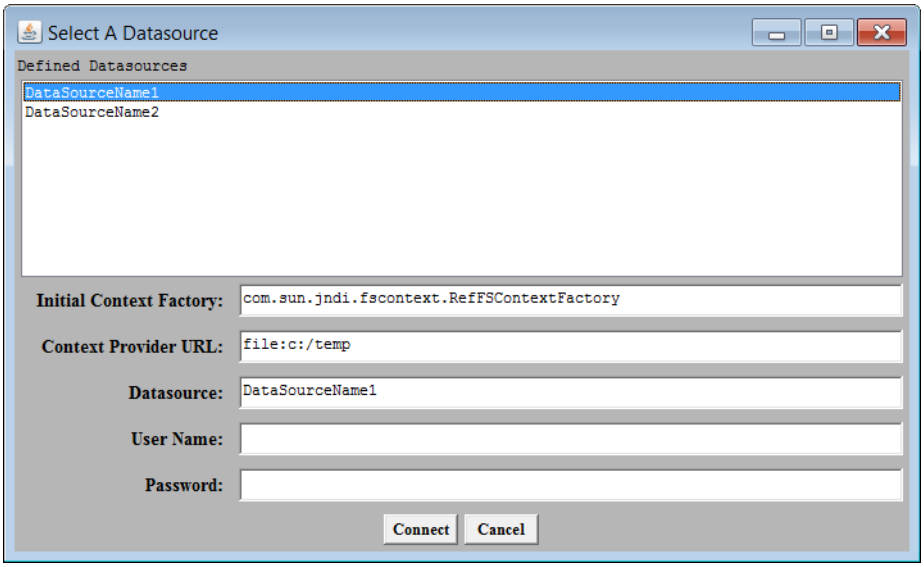
3. Click **Press Here to Continue**.

   The main dialog appears:



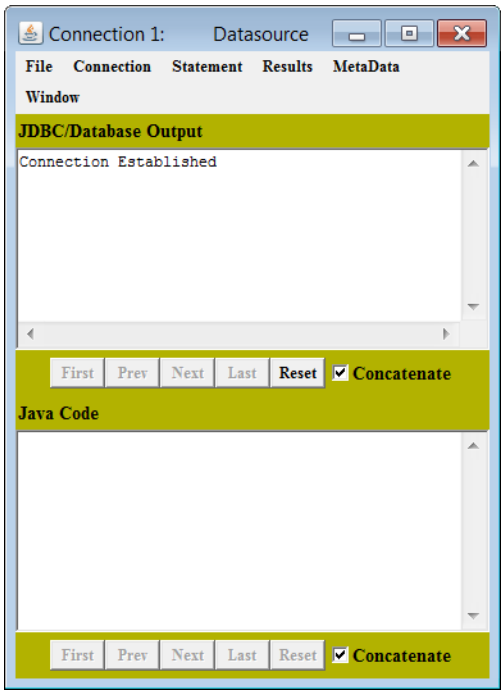
4. From the menu bar, select **Connection > Connect to DB via Data Source**.

   The **Select A Database** dialog appears:



5. Select a datasource template from the **Defined Datasources** field.
6. Provide the following information:
   a) In the **Initial Context Factory**, specify the location of the initial context provider for your application.
   b) In the **Context Provider URL**, specify the location of the context provider for your application.
   c) In the **Datasource** field, specify the name of your datasource.
7. If you are using user ID/password authentication, enter your user ID and password in the corresponding fields.
8. Click **Connect**.

If the connection information is entered correctly, the **JDBC/Database Output** window reports that a connection has been established. If a connection is not established, the window reports an error.



Authentication

The driver supports the user ID/password authentication. It authenticates using a database user name and password provided by the application.

To configure the driver to use user ID/password authentication.

- Set the ServerName property to specify either the IP address in IPv4 or IPv6 format, or the name of your server.
- Set the PortNumber property to specify the TCP port of the primary database server that is listening for connections to the database.

  Note: The driver connects to the 9996 port and any customized port that is configured on the PostgreSQL wire protocol.

- Set the DatabaseName property to specify the name of the database to which you want to connect.
- Set the User property to specify the user name that is used to connect to the server.
- Set the Password property to specify the password.

The following examples show the connection information required to establish a connection using user ID/password authentication.

Connection URL

```
Connection conn = DriverManager.getConnection
("jdbc:datadirect:denodo://myserver:9996;
  DatabaseName=mydb;User=jsmith;Password=secret);
```

Data Source

```
DenodoDataSource mds = new DenodoDataSource();
mds.setDescription("My Denodo Data Source");
mds.setServerName("myserver");
mds.setPortNumber("9996");
mds.setDatabaseName("payroll");
mds.setUser("jsmith");
mds.setPassword("secret");
```

Note: The User and Password properties are not required to be stored in the connection string. They can also be passed separately by the application.

Note: Setting the password using a data source is generally not recommended. The data source persists all properties, including the Password property, in clear text.

See also

User on page 102
Password on page 90

Data Encryption

TLS/SSL works by allowing the client and server to send each other encrypted data that only they can decrypt. TLS/SSL negotiates the terms of the encryption in a sequence of events known as the *handshake*. The handshake involves the following types of authentication:

- *TLS/SSL server authentication* requires the server to authenticate itself to the client.
- *TLS/SSL client authentication* is optional and requires the client to authenticate itself to the server after the server has authenticated itself to the client.

Configuring TLS/SSL Encryption

The driver supports TLS/SSL encryption for Denodo.

Note: Connection hangs can occur when the driver is configured for TLS/SSL and the database server does not support TLS/SSL. You may want to set a login timeout using the LoginTimeout property to avoid problems when connecting to a server that does not support TLS/SSL.

To configure TLS/SSL encryption:

- Set the ServerName property to the name or the IP address of the Denodo server to which you want to connect. For example, `myserver`.
- Set the PortNumber property to specify the port number of the server listener. The default is `9996`.

- Set the DatabaseName property to the name of the database to which you want to connect.
- Set the User property to specify the user name that is used to connect to the server.
- Set the Password property to specify the password.
- Set the EncryptionMethod property to SSL.
- Specify the location and password of the truststore file used for SSL server authentication. Either set the TrustStore and TrustStorePassword properties or their corresponding Java system properties (javax.net.ssl.trustStore and javax.net.ssl.trustStorePassword, respectively).
- (Optional) Set the CryptoProtocolVersion property to specify acceptable cryptographic protocol versions (for example, TLSv1.2) supported by your server.
- (Optional) To validate certificates sent by the database server, set the ValidateServerCertificate property to true.
- (Optional) Set the HostNameInCertificate property to a host name to be used to validate the certificate. The HostNameInCertificate property provides additional security against man-in-the-middle (MITM) attacks by ensuring that the server the driver is connecting to is the server that was requested.
- (Optional) If your database server is configured for SSL client authentication, configure your keystore information:
  - Specify the location and password of the keystore file. Either set the KeyStore and KeyStorePassword properties or their corresponding Java system properties (javax.net.ssl.keyStore and javax.net.ssl.keyStorePassword, respectively).
  - If any key entry in the keystore file is password-protected, set the KeyPassword property to the key password.

Note: The User and Password properties are not required to be stored in the connection string. They can also be passed separately by the application.

The following examples demonstrate the required properties for a session using TLS/SSL encryption with user ID/password authentication.

For a connection URL:

```
Connection conn = DriverManager.getConnection
 ("jdbc:datadirect:denodo://myserver:9996;DatabaseName=mydb;
   User=jsmith;Password=secret;EncryptionMethod=SSL
   TrustStore=TrustStoreFile;TrustStorePassword=XYZ;");
```

For a data source:

```
DenodoDataSource mds = new DenodoDataSource();
mds.setDescription("My Denodo Data Source");
mds.setServerName("myserver");
mds.setPortNumber("9996");
mds.setDatabaseName("mydb");
mds.setUser("jsmith");
mds.setPassword("secret");
mds.setEncryptionMethod("SSL");
mds.setTrustStore("TrustStoreFile");
mds.setTrustStorePassword("XYZ");
```

Note: Setting the password using a data source is generally not recommended. The data source persists all properties, including the Password property, in clear text.

See also

Connection property descriptions on page 57
Configuring TLS/SSL Server Authentication on page 44
Configuring TLS/SSL Client Authentication on page 44

Configuring TLS/SSL Server Authentication

When the client makes a connection request, the server presents its public certificate for the client to accept or deny. The client checks the issuer of the certificate against a list of trusted Certificate Authorities (CAs) that resides in an encrypted file on the client known as a *truststore*. Optionally, the client may check the subject (owner) of the certificate. If the certificate matches a trusted CA in the truststore (and the certificate's subject matches the value that the application expects), an encrypted connection is established between the client and server. If the certificate does not match, the connection fails and the driver throws an exception.

To check the issuer of the certificate against the contents of the truststore, the driver must be able to locate the truststore and unlock the truststore with the appropriate password. You can specify truststore information in either of the following ways:

- Specify values for the Java system properties javax.net.ssl.trustStore and javax.net.ssl.trustStorePassword. For example:

  ```
  java -Djavax.net.ssl.trustStore=C:\Certificates\MyTruststore
       -Djavax.net.ssl.trustStorePassword=MyTruststorePassword
  ```

  This method sets values for all TLS/SSL sockets created in the JVM.

- Specify values for the connection properties TrustStore and TrustStorePassword in the connection URL. For example:

  ```
  TrustStore=C:\Certficates\MyTruststore
  ```

  and

  ```
  TrustStorePassword=MyTruststorePassword
  ```

  Any values specified by the TrustStore and TrustStorePassword properties override values specified by the Java system properties. This allows you to choose which truststore file you want to use for a particular connection.

Alternatively, you can configure the drivers to trust any certificate sent by the server, even if the issuer is not a trusted CA. Allowing a driver to trust any certificate sent from the server is useful in test environments because it eliminates the need to specify truststore information on each client in the test environment. If the driver is configured to trust any certificate sent from the server, the issuer information in the certificate is ignored.

Configuring TLS/SSL Client Authentication

If the server is configured for TLS/SSL client authentication, the server asks the client to verify its identity after the server has proved its identity. Similar to TLS/SSL server authentication, the client sends a public certificate to the server to accept or deny. The client stores its public certificate in an encrypted file known as a *keystore*.

The driver must be able to locate the keystore and unlock the keystore with the appropriate keystore password. Depending on the type of keystore used, the driver also may need to unlock the keystore entry with a password to gain access to the certificate and its private key.

The drivers can use the following types of keystores:

- Java Keystore (JKS) contains a collection of certificates. Each entry is identified by an alias. The value of each entry is a certificate and the certificate's private key. Each keystore entry can have the same password as the keystore password or a different password. If a keystore entry has a password different than the keystore password, the driver must provide this password to unlock the entry and gain access to the certificate and its private key.
- PKCS #12 keystores. To gain access to the certificate and its private key, the driver must provide the keystore password. The file extension of the keystore must be .pfx or .p12.

You can specify this information in either of the following ways:

- Specify values for the Java system properties javax.net.ssl.keyStore and javax.net.ssl.keyStorePassword. For example:

```
java -Djavax.net.ssl.keyStore=C:\Certificates\MyKeystore
     -Djavax.net.ssl.keyStorePassword=MyKeystorePassword
```

  This method sets values for all TLS/SSL sockets created in the JVM.

Note: If the keystore specified by the javax.net.ssl.keyStore Java system property is a JKS and the keystore entry has a password different than the keystore password, the KeyPassword connection property must specify the password of the keystore entry (for example, `KeyPassword=MyKeyPassword`).

- Specify values for the connection properties KeyStore and KeyStorePassword in the connection URL. For example:

  `KeyStore=C:\Certficates\MyKeyStore`

  and

  `KeyStorePassword=MyKeystorePassword`

Note: If the keystore specified by the KeyStore connection property is a JKS and the keystore entry has a password different than the keystore password, the KeyPassword connection property must specify the password of the keystore entry (for example, `KeyPassword=MyKeyPassword`).

Any values specified by the KeyStore and KeyStorePassword properties override values specified by the Java system properties. This allows you to choose which keystore file you want to use for a particular connection.

Failover

The driver provides connection failover support to ensure continuous, uninterrupted access to data. It allows you to specify a list of alternate database servers that are tried at connection time if the primary server is not accepting connections. Connection attempts continue until a connection is successfully established or until all the database servers in the list have been tried the specified number of times.

Note: For general information on failover, refer to "Failover" in the *Progress DataDirect for JDBC Drivers Reference*.

To configure failover:

1. Specify the primary and alternate servers:

a. Specify your primary server using a connection URL or data source.

b. Specify one or multiple alternate servers by setting the AlternateServers property.

Note: To turn off failover, do not specify a value for the AlternateServers property.

2. Optionally, configure the connection retry feature by setting the ConnectionRetryCount and ConnectionRetryDelay connection properties.
3. Optionally, set the LoadBalancing property to determine whether the driver attempts to connect to the database servers (primary and alternate) in a random order or a sequential order.

The following examples configure the driver to use connection failover in conjunction with connection retry.

Note: The User and Password properties are not required to be stored in the connection string. They can also be passed separately by the application.

Connection URL

```
jdbc:datadirect:denodo://myserver1:9996;
DatabaseName=mydb1;User=jsmith;Password=secret;
AlternateServers=(myserver2:9996;DatabaseName=mydb2,myserver3:9996;DatabaseName=mydb3);
ConnectionRetryCount=2;ConnectionRetryDelay=5
```

In this example:

```
...myserver1:9996;DatabaseName=mydb1...
```

is the part of the connection URL that specifies connection information for the primary server. Alternate servers are specified using the AlternateServers property. For example:

```
...AlternateServers=(myserver2:9996;DatabaseName=mydb2,myserver3:9996;DatabaseName=mydb3)
```

If you do not specify the DatabaseName connection property in an alternate server entry, the connection to that alternate server uses the property specified in the URL for the primary server. For example, if you specify `DatabaseName=mydb1` for the primary server, but do not specify a database name in the alternate server entry, the driver tries to connect to the mydb1 database on the alternate server.

If a successful connection is not established on the Denodo driver's first pass through the list of database servers (primary and alternate), the driver retries the list of servers in the same sequence twice (`ConnectionRetryCount=2`). Because the connection retry delay has been set to five seconds (`ConnectionRetryDelay=5`), the driver waits five seconds between retry passes.

Data Source

```
DenodoDataSource mds = new DenodoDataSource();
mds.setDescription("My Denodo Data Source");
mds.setServerName("myserver1");
mds.setPortNumber("9996");
mds.setDatabaseName("mydb1");
mds.setUser("jsmith");
mds.setPassword("secret");
mds.setAlternateServers("myserver2:9996;DatabaseName=mydb2,
                         myserver3:9996,DatabaseName=mydb3");
```

Note: Setting the password using a data source is generally not recommended. The data source persists all properties, including the Password property, in clear text.

See also

Connection property descriptions on page 57

Using client information

Many databases allow applications to store client information associated with a connection, which can be useful for database administration and monitoring purposes. The driver allows applications to store and return the following types of client information.

- Name of the application currently using the connection.
- User ID for whom the application using the connection is performing work. The user ID may be different than the user ID that was used to establish the connection.
- Host name of the client on which the application using the connection is running.
- Product name and version of the driver on the client.
- Additional information that may be used for accounting or troubleshooting purposes, such as an accounting ID.

Refer to "Client information" in the *Progress DataDirect for JDBC Drivers Reference* for more information.

Bulk load

As Denodo does not have native bulk load support, the Denodo driver emulates bulk load using the standard batch mechanism.

The BulkLoadBatchSize connection property affects how bulk load works with the Denodo driver. It suggests the number of rows to be loaded to the database when bulk loading data. The default is `1000`.

See also

BulkLoadBatchSize on page 68

Performance considerations

You can optimize application performance by adopting guidelines described in this section.

BulkLoadBatchSize: The BulkLoadBatchSize property is used to specify the number of rows the driver loads at a time when bulk loading data. Performance can be improved by increasing the number of rows because fewer network round trips are required. For example, if BulkLoadBatchSize is set to 10,000 rows and you are inserting 100,000 rows, the driver executes 10 batch inserts that require separate round trips to complete the bulk operation. Be aware that increasing the number of rows that are loaded also causes the driver to consume more memory on the client.

EncryptionMethod: Data encryption may adversely affect performance because of the additional overhead (mainly CPU usage) required to encrypt and decrypt data.

InsensitiveResultSetBufferSize: To improve performance when using scroll-insensitive result sets, the driver can cache the result set data in memory instead of writing it to disk. By default, the driver caches 2 MB of insensitive result set data in memory and writes any remaining result set data to disk. Performance can be improved by increasing the amount of memory used by the driver before writing data to disk or by forcing the driver to never write insensitive result set data to disk. The maximum cache size setting is 2 GB.

MaxPooledStatements: To improve performance, the driver's own internal prepared statement pooling should be enabled when the driver does not run from within an application server or from within another application that does not provide its own prepared statement pooling. When the driver's internal prepared statement pooling is enabled, the driver caches a certain number of prepared statements created by an application. For example, if the MaxPooledStatements property is set to `20`, the driver caches the last 20 prepared statements created by the application. If the value set for this property is greater than the number of prepared statements used by the application, all prepared statements are cached.

Refer to "Designing JDBC Applications for Performance Optimization" in the *Progress DataDirect for JDBC Drivers Reference* for more information about using prepared statement pooling to optimize performance.

ResultSetMetaDataOptions: The driver’s performance may be adversely affected if you set this option to 1. If set to 1 and the ResultSetMetaData.getTableName method is called, the driver performs emulations which take additional processing.

VarcharClobThreshold: There are performance penalties when enabling CLOB functionality. To provide the benefits associated with Clobs, data must be cached. Because data is cached, your application will incur a performance penalty, particularly if data is read once sequentially. This performance penalty can be severe if the size of the long data is larger than available memory. If you want to avoid the performance penalties associated with CLOB functionality, you should set this value at a value greater than the maximum Character varying column width your application handles.

4

Additional features and functionality

The following section describes additionally supported features and functionality that are specific to the driver.

For details, see the following topics:

- Connection pooling
- Statement pooling
- Isolation levels
- Returning and inserting/updating XML data
- Using scrollable cursors
- JTA support
- Batch inserts
- Large object (LOB) support
- Parameter metadata support
- ResultSet metadata support
- Rowset support

Connection pooling

Progess DataDirect for JDBC drivers support connection pooling using the DataDirect Connection Pool Manager. Typically, creating a connection is the most performance-expensive operations that an application performs. Connection pooling allows you to reuse connections rather than create a new one every time a driver needs to connect to the database. Further, connection pooling manages connection sharing across multiple user requests to maintain performance and reduce the number of new connections to be created.

Refer to "Connection Pool Manager" in the *Progress DataDirect for JDBC Drivers Reference* for more information.

Statement pooling

Most applications have a certain set of SQL statements that are executed multiple times and a few SQL statements that are executed only once or twice during the life of the application. Similar to connection pooling, *statement pooling* provides performance gains for applications that execute the same SQL statements multiple times over the life of the application.

A *statement pool* is a group of prepared statements that can be reused by an application. If you have an application that repeatedly executes the exact same SQL statements, statement pooling can improve performance because the database server does not have to repeatedly parse and create cursors for the same statement. In addition, the associated network round trips to the database server are avoided.

The drivers have an internal prepared statement pooling mechanism, which allows you to realize the performance benefits of statement pooling when you are not running from within an application server or another application that provides its own statement pooling. You can enable this driver-based internal statement pooling with the MaxPooledStatements connection property.

The DataDirect for JDBC drivers also support the DataDirect Statement Pool Monitor. You can use the Statement Pool Monitor to load statements into and remove statements from the statement pool as well as generate information to help you troubleshoot statement pooling performance. The Statement Pool Monitor is an integrated component of the driver, and you can manage statement pooling directly with DataDirect-specific methods. In addition, the Statement Pool Monitor can be enabled as a Java Management Extensions (JMX) MBean. When enabled as a JMX MBean, the Statement Pool Monitor can be used to manage statement pooling with standard JMX API calls, and it can easily be used by JMX-compliant tools, such as JConsole. To enable the Statement Pool Monitor as a JMX MBean, you must register the Statement Pool Monitor MBean with the RegisterStatementPoolMonitorMBean connection property.

Refer to "Statement Pool Monitor" in the *Progress DataDirect for JDBC Drivers Reference* for further details.

See also

MaxPooledStatements on page 88
RegisterStatementPoolMonitorMBean on page 96

Isolation levels

The driver supports the Read Committed, Read Uncommitted, Repeatable Read, and Serializable isolation levels. The default is Read Committed.

Returning and inserting/updating XML data

The driver supports the XML data type for Denodo.

Returning XML data

The driver returns XML data as character data. Your application can use the following methods to return data stored in XML columns as character data:

- ResultSet.getString()
- ResultSet.getCharacterStream()
- ResultSet.getClob()
- CallableStatement.getString()
- CallableStatement.getClob()

The driver converts the XML data returned from the database server from the UTF-8 encoding used by the database server to the UTF-16 Java String encoding.

Your application can use the following method to return data stored in XML columns as ASCII data:

- ResultSet.getAsciiStream()

The driver converts the XML data returned from the database server from the UTF-8 encoding to the ISO-8859-1 (latin1) encoding.

Note: The conversion caused by using the getAsciiStream() method may create XML that is not well-formed because the content encoding is not the default encoding and does not contain an XML declaration specifying the content encoding. Do not use the getAsciiStream() method if your application requires well-formed XML.

Inserting/updating XML data

The driver can insert or update XML data as character or binary data.

Character data

Your application can use the following methods to insert or update XML data as character data:

- PreparedStatement.setString()
- PreparedStatement.setCharacterStream()
- PreparedStatement.setClob()
- PreparedStatement.setObject()
- ResultSet.updateString()
- ResultSet.updateCharacterStream()
- ResultSet.updateClob()

- ReultSet.updateObject()

The driver converts the character representation of the data to the XML character set used by the database server and sends the converted XML data to the server. The driver does not parse or remove any XML processing instructions.

Your application can update XML data as ASCII data using the following methods:

- PreparedStatement.setAsciiStream()
- ResultSet.updateAsciiStream()

The driver interprets the data supplied to these methods using the ISO-8859-1 (latin 1) encoding. The driver converts the data from ISO-8859-1 to the XML character set used by the database server and sends the converted XML data to the server.

Binary data

Your application can use the following methods to insert or update XML data as binary data:

- PreparedStatement.setBytes()
- PreparedStatement.setBinaryStream()
- PreparedStatement.setBlob()
- PreparedStatement.setObject()
- ResultSet.updateBytes()
- ResultSet.updateBinaryStream()
- ResultSet.updateBlob()
- ReultSet.updateObject()

The driver does not apply any data conversions when sending XML data to the database server.

Using scrollable cursors

The driver supports forward-only and scroll-insensitive results.

Note: When the driver cannot support the requested result set type or concurrency, it automatically downgrades the cursor and generates one or more SQLWarnings with detailed information.

JTA support

JDBC distributed transactions through JTA are not supported by the driver.

Batch inserts

The BatchMechanism connection property determines how the driver manages batch inserts. When BatchMechanism is set to `nativeBatch`, the driver uses the Denodo native batch protocol to insert all batched parameters. See "BatchMechanism" for details.

See also

BatchMechanism on page 67

Large object (LOB) support

The Denodo driver allows you to retrieve and update long data, specifically LONGVARBINARY and LONGVARCHAR [1] data, using JDBC methods designed for Blobs and Clobs. When using these methods to update long data as Blobs or Clobs, the updates are made to the local copy of the data contained in the Blob or Clob object.

Retrieving and updating long data using JDBC methods designed for Blobs and Clobs provides some of the same benefits as retrieving and updating Blobs and Clobs, such as:

- Provides random access to data
- Allows searching for patterns in the data, such as retrieving long data that begins with a specific character string

To provide these benefits normally associated with Blobs and Clobs, data must be cached. Because data is cached, your application will incur a performance penalty, particularly if data is read once sequentially. This performance penalty can be severe if the size of the long data is larger than available memory.

Parameter metadata support

The driver supports returning parameter metadata. The driver returns only the parameter's data type.

User-defined function results

Denodo provides functionality to create user-defined functions. Denodo does not define a call mechanism for invoking a user-defined function. User-defined functions must be invoked via a SQL statement. For example, given a function defined as:

```
CREATE table foo (intcol int, varcharcol varchar(123))
CREATE or REPLACE FUNCTION insertFoo
  (IN idVal int, IN nameVal varchar) RETURNS void
  AS $$
    insert into foo values ($1, $2);
  $$
  LANGUAGE SQL;
```

[1] You may determine whether *Character varying* columns are described as VARCHAR or LONGVARCHAR by setting the VarcharClobThreshold on page 103 connection property.

must be invoked natively as:

```
SELECT * FROM insertFoo(100, 'Mark')
```

even though the function does not return a value or results. The Select SQL statement returns a result set that has one column named insertFoo and no row data.

The Denodo driver supports invoking user-defined functions using the JDBC call Escape. The previously described function can be invoked using:

```
{call insertFoo(100, 'Mark')}
```

Denodo functions return data from functions as a result set. If multiple output parameters are specified, the values for the output parameters are returned as columns in the result set. For example, the function defined as:

```
CREATE or REPLACE FUNCTION addValues(in v1 int, in v2 int)
  RETURNS int
  AS $$
    SELECT $1 + $2;
  $$
  LANGUAGE SQL;
```

returns a result set with a single column of type INTEGER, whereas the function defined as:

```
CREATE or REPLACE FUNCTION selectFooRow2
  (IN idVal int, OUT id int, OUT name varchar)
  AS $$
    select intcol, varcharcol from foo where intcol = $1;
  $$
  LANGUAGE SQL
```

returns a result set that contains two columns, a INTEGER id column and a VARCHAR name column.

In addition, when calling Denodo functions that contain output parameters, the native syntax requires that the output parameter values be omitted from the function call. This, in addition to output parameter values being returned as a result set, makes the Denodo behavior of calling functions different from most other databases.

Note: When using the ?= version of the call escape on a function that returns a set of values, only the first result in the set will be returned.

The Denodo driver provides a mechanism that makes the invoking of functions more consistent with how other databases behave. In particular, the Denodo driver allows parameter markers for output parameters to be specified in the function argument list when the Escape call is used. The driver extracts the output parameter values from the result set returned by the server and makes the values available via the getxxx methods on a CallableStatement.

For example, the function `selectFooRow2` described previously can be invoked as:

```
sql = "{call selectFooRow2 (?,?,?)}";
CallableStatement cSTMT = connection.prepareCall(sql);
cSTMT.setInt(1, idVal);
cSTMT.registerOutParameter(2, Types.INTEGER);
cSTMT.registerOutParameter(3, Types.VARCHAR);
cSTMT.execute();
int myID = cSTMT.getInt(2);
String myName = cSTMT.getString(3);
```

The values of the id and name output parameters are returned in the `myID` and `myName` variables.

An error is returned if the number of output parameters registered when the function is executed is less than the number of output parameters defined in the function. If no output parameters are registered to a function call, the driver returns the output parameters as a result set.

ResultSet metadata support

If your application requires table name information, the driver can return table name information in ResultSet metadata for Select statements. If you set the ResultSetMetaDataOptions property to 1, the driver performs additional processing to determine the correct table name for each column in the result set when the ResultSetMetaData.getTableName() method is called. Otherwise, the getTableName() method may return an empty string for each column in the result set.

When the ResultSetMetaDataOptions property is set to `1` and the ResultSetMetaData.getTableName() method is called, the table name information that is returned by the driver depends on whether the column in a result set maps to a column in a table in the database. For each column in a result set that maps to a column in a table in the database, the driver returns the table name associated with that column. For columns in a result set that do not map to a column in a table (for example, aggregates and literals), the driver returns an empty string.

The Select statements for which ResultSet metadata is returned may contain aliases, joins, and fully qualified names. The following queries are examples of Select statements for which the ResultSetMetaData.getTableName() method returns the correct table name for columns in the Select list:

```
SELECT id, name FROM Employee
SELECT E.id, E.name FROM Employee E
SELECT E.id, E.name AS EmployeeName FROM Employee E
SELECT E.id, E.name, I.location, I.phone FROM Employee E, EmployeeInfo I
   WHERE E.id = I.id
SELECT id, name, location, phone FROM Employee, EmployeeInfo WHERE id = empId
SELECT Employee.id, Employee.name, EmployeeInfo.location, EmployeeInfo.phone
   FROM Employee, EmployeeInfo WHERE Employee.id = EmployeeInfo.id
```

The table name returned by the driver for generated columns is an empty string. The following query is an example of a Select statement that returns a result set that contains a generated column (the column named "upper").

```
SELECT E.id, E.name as EmployeeName, {fn UCASE(E.name)} AS upper FROM Employee E
```

The driver also can return catalog name information when the ResultSetMetaData.getCatalogName() method is called if the driver can determine that information. For example, for the following statement, the driver returns "test" for the catalog name and "foo" for the table name:

```
SELECT * FROM test.foo
```

The additional processing required to return table name and catalog name information is only performed if the ResultSetMetaData.getTableName() or ResultSetMetaData.getCatalogName() methods are called.

Rowset support

The driver supports any JSR 114 implementation of the RowSet interface, including:

- CachedRowSets
- FilteredRowSets
- WebRowSets
- JoinRowSets
- JDBCRowSets

Visit https://www.jcp.org/en/jsr/detail?id=114 for more information about JSR 114.

5

Connection property descriptions

You can use connection properties to customize the driver for your environment. This section organizes connection properties according to functionality. You can use connection properties with either the JDBC `DriverManager` or a JDBC data source. For a `DriverManager` connection, a property is expressed as a key value pair and takes the form *`property=value`*. For a data source connection, a property is expressed as a JDBC method and takes the form `set`*`property`*`(`*`value`*`)`.

Note:

- In a JDBC data source, string values must be enclosed in double quotation marks, for example, `setUser("abc@defcorp.com")`.
- The data type listed for each connection property is the Java data type used for the property value in a JDBC data source.
- Connection property names are case-insensitive. For example, `Password` is the same as `password`.
- For connection properties that support string values, use the following escape sequence to specify values containing leading or trailing spaces and curly brackets: `{`*`value`*`}`. For example: `User={hello }` or `Password={{hello}}`.

The following tables describe the connection properties by functionality.

- Timeout properties
- Client information properties
- Statement pooling properties
- Additional properties

User ID/password authentication properties

The following table summarizes the connection properties required for user ID/password authentication.

Property	Data Source Method	Default
User on page 102	`setUser` `getUser`	No default value
Password on page 90	`setPassword` `getPassword`	No default value
ServerName on page 98	`setServerName` `getServerName`	No default value
PortNumber on page 90	`setPortNumber` `getPortNumber`	`9996`
DatabaseName on page 75	`setDatabaseName` `getDatabaseName`	No default value

Proxy server properties

The following table summarizes the proxy server connection properties.

Property	Data Source Method	Default
ProxyHost on page 93	`getProxyHost()` `setProxyHost(String)`	No default value
ProxyPassword on page 93	`getProxyPassword()` `setProxyPassword(String)`	No default value

Property	Data Source Method	Default
ProxyPort on page 94	`getProxyPort()` `setProxyPort(Integer)`	`0` which means the default is determined by the ProxyHost property. For HTTP URLs: `80` For HTTPS URLs: `443`
ProxyUser on page 95	`getProxyUser()` `setProxyUser(String)`	No default value

Data encryption properties

The following table summarizes connection properties that can be used to enable SSL.

Property	Data Source Method	Default
CryptoProtocolVersion on page 74	`setCryptoProtocolVersion` `getCryptoProtocolVersion`	No default value
EncryptionMethod on page 77	`setEncryptionMethod` `getEncryptionMethod`	`noEncryption`
HostNameInCertificate on page 78	`setHostNameInCertificate` `getHostNameInCertificate`	No default value
KeyStore on page 83	`setKeyStore` `getKeyStore`	No default value
KeyStorePassword on page 84	`setKeyStorePassword` `getKeyStorePassword`	No default value
KeyPassword on page 82	`setKeyPassword` `getKeyPassword`	No default value
TrustStore on page 101	`setTrustStore` `getTrustStore`	No default value
TrustStorePassword on page 101	`setTrustStorePassword` `getTrustStorePassword`	No default value
ValidateServerCertificate on page 103	`setValidateServerCertificate` `getValidateServerCertificate`	`true`

Bulk load properties

The following table contains the only connection property that affects how bulk load works with the driver.

Property	Data Source Method	Default
BulkLoadBatchSize on page 68	setBulkLoadBatchSize getBulkLoadBatchSize	1000

Failover properties

The following table summarizes the connection properties used for configuring failover.

Property	Data Source Method	Default
AlternateServers on page 65	setAlternateServers getAlternateServers	No default value
ConnectionRetryCount on page 71	setConnectionRetryCount getConnectionRetryCount	5
ConnectionRetryDelay on page 72	setConnectionRetryDelay getConnectionRetryDelay	1 (second)
DatabaseName on page 75	setDatabaseName getDatabaseName	No default value
LoadBalancing on page 85	setLoadBalancing getLoadBalancing	false

Timeout properties

The following table summarizes timeout connection properties.

Property	Data Source Method	Default
EnableCancelTimeout on page 75	setEnableCancelTimeout getEnableCancelTimeout	false
LoginTimeout on page 85	setLoginTimeout getLoginTimeout	0
QueryTimeout on page 95	setQueryTimeout getQueryTimeout	0

Client information properties

The following table summarizes connection properties that can be used to return client information.

Property	Data Source Method	Default
AccountingInfo on page 65	setAccountingInfo getAccountingInfo	No default value
ApplicationName on page 66	setApplicationName getApplicationName	No default value
ClientHostName on page 70	setClientHostName getClientHostName	No default value
ClientUser on page 71	setClientUser getClientUser	No default value
ProgramID on page 92	setProgramID getProgramID	No default value

Statement pooling properties

The following table summarizes statement pooling connection properties.

Property	Data Source Method	Default
ImportStatementPool on page 79	setImportStatementPool getImportStatementPool	No default value
MaxPooledStatements on page 88	setMaxPooledStatements getMaxPooledStatements	0
RegisterStatementPoolMonitorMBean on page 96	setRegisterStatementPoolMonitorMBean getRegisterStatementPoolMonitorMBean	false

Additional properties

The following table summarizes additional connection properties.

Property	Data Source Method	Default
BatchMechanism on page 67	setBatchMechanism getBatchMechanism	nativeBatch
CallEscapeBehavior on page 69	setCallEscapeBehavior getCallEscapeBehavior	callIfNoReturn

Property	Data Source Method	Default
CatalogOptions on page 69	setCatalogOptions getCatalogOptions	2
ConvertNull on page 73	setConvertNull getConvertNull	1
EnablePrepareThreshold on page 76	setEnablePrepareThreshold getEnablePrepareThreshold	true
ExtendedColumnMetadata on page 78	setExtendedColumnMetadata getExtendedColumnMetadata	false
InitializationString on page 80	setInitializationString getInitializationString	No default value
InsensitiveResultSetBufferSize on page 81	setInsensitiveResultSetBufferSize getInsensitiveResultSetBufferSize	2048
JavaDoubleToString on page 82	setJavaDoubleToString getJavaDoubleToString	false
MaxLongVarcharSize on page 86	setMaxLongVarcharSize getMaxLongVarcharSize	1,073,741,823
MaxNumericPrecision on page 87	setMaxNumericPrecision getMaxNumericPrecision	1000
MaxNumericScale on page 87	setMaxNumericScale getMaxNumericScale	998
MaxStatements on page 89	setMaxStatements getMaxStatements	0
MaxVarcharSize on page 89	setMaxVarcharSize getMaxVarcharSize	10,485,760
PrepareThreshold on page 91	setPrepareThreshold getPrepareThreshold	0
ResultSetMetaDataOptions on page 97	setResultSetMetaDataOptions getResultSetMetaDataOptions	0

Property	Data Source Method	Default
SpyAttributes on page 98	`setSpyAttributes` `getSpyAttributes`	No default value
SupportsCatalogs on page 99	`setSupportsCatalogs` `getSupportsCatalogs`	`true`
TransactionErrorBehavior on page 100	`setTransactionErrorBehavior` `getTransactionErrorBehavior`	`RollbackTransaction`
VarcharClobThreshold on page 103	`setVarcharClobThreshold` `getVarcharClobThreshold`	`32768`

For details, see the following topics:

- AccountingInfo
- AlternateServers
- ApplicationName
- BatchMechanism
- BulkLoadBatchSize
- CallEscapeBehavior
- CatalogOptions
- ClientHostName
- ClientUser
- ConnectionRetryCount
- ConnectionRetryDelay
- ConvertNull
- CryptoProtocolVersion
- DatabaseName
- EnableCancelTimeout
- EnablePrepareThreshold
- EncryptionMethod
- ExtendedColumnMetadata
- HostNameInCertificate
- ImportStatementPool
- InitializationString

- InsensitiveResultSetBufferSize
- JavaDoubleToString
- KeyPassword
- KeyStore
- KeyStorePassword
- LoadBalancing
- LoginTimeout
- MaxLongVarcharSize
- MaxNumericPrecision
- MaxNumericScale
- MaxPooledStatements
- MaxStatements
- MaxVarcharSize
- Password
- PortNumber
- PrepareThreshold
- ProgramID
- ProxyHost
- ProxyPassword
- ProxyPort
- ProxyUser
- QueryTimeout
- RegisterStatementPoolMonitorMBean
- ResultSetMetaDataOptions
- ServerName
- SpyAttributes
- SupportsCatalogs
- TransactionErrorBehavior
- TrustStore
- TrustStorePassword
- User
- ValidateServerCertificate
- VarcharClobThreshold

AccountingInfo

Purpose

Defines accounting information. This value is stored locally and is used for database administration/monitoring purposes.

Valid Values

string

where:

string

is the accounting information.

Data Source Methods

```
public String getAccountingInfo()
```

```
public void setAccountingInfo(String)
```

Default

No default value

Data Type

String

See also

Using client information on page 47

AlternateServers

Purpose

Defines a list of alternate database servers that is used to failover new or lost connections, depending on the failover method selected.

Valid Values

```
(servername1[:port1][;DatabaseName=value][,servername2[:port2]
[;DatabaseName=value]]...)
```

The server name (*servername1*, *servername2*, and so on) is required for each alternate server entry. Port number (*port1*, *port2*, and so on) and connection properties (*property*=*value*) are optional for each alternate server entry. If the port is unspecified, the port number of the primary server is used. If the port number of the primary server is unspecified, the default port number of 9996 is used.

DatabaseName is an optional connection property.

Data Source Methods

```
public String getAlternateServers()
```

```
public void setAlternateServers(String)
```

Default

No default value

Data Type

String

Example

The following URL contains alternate server entries for server2 and server3. The alternate server entries contain the optional DatabaseName property.

```
jdbc:datadirect:denodo://server1:9996;DatabaseName=TEST;User=test;
Password=secret;AlternateServers=(server2:9996;DatabaseName=TEST2,
server3:9996;DatabaseName=TEST3)
```

Note: The User and Password properties are not required to be stored in the connection string. They can also be passed separately by the application.

See also

Failover on page 45

ApplicationName

Purpose

Specifies the name of the application. This value is stored locally and is used for database administration/monitoring purposes.

Valid Values

string

where:

string

is the name of the application.

Data Source Methods

```
public String getApplicationName()
```

```
public void setApplicationName(String)
```

Default

No default value

Data Type

String

See also

Using client information on page 47

BatchMechanism

Purpose

Determines the mechanism that is used to execute batch inserts.

Valid Values

`nativeBatch`

Behavior

If set to `nativeBatch`, the driver uses the Denodo native batch protocol to insert all batched parameters.

Notes

- BatchMechanism determines the mechanism used to perform batch inserts only. For update and delete batch operations, the driver uses the native batch mechanism to handle the request.
- When `BatchMechanism=nativeBatch`, individual update counts are returned for each statement or parameter set in the batch as required by the JDBC 3.0 specification.

Data Source Methods

```
public String getBatchMechanism()
```

```
public void setBatchMechanism(String)
```

Default

`nativeBatch`

Data Type

String

See also

- Performance considerations on page 47
- Batch inserts on page 53

BulkLoadBatchSize

Purpose

Provides a suggestion to the driver for the number of rows to load to the database at a time when bulk loading data. Performance can be improved by increasing the number of rows the driver loads at a time because fewer network round trips are required. Be aware that increasing the number of rows that are loaded also causes the driver to consume more memory on the client.

Valid Values

x

where:

x

is a positive integer that represents a number of rows.

Notes

- This property suggests the number of rows regardless of which bulk load method is used: using a DDBulkLoad object or using bulk load for batch inserts.
- The DDBulkObject.setBatchSize() method overrides the value that is set by this property.

  Refer to "JDBC extensions" in the *Progress DataDirect for JDBC Drivers Reference* for more information about bulk load methods.

Data Source Methods

```
public Long getBulkLoadBatchSize()
```

```
public void setBulkLoadBatchSize(Long)
```

Default

```
1000
```

Data Type

long

See also

Bulk load on page 47

Performance considerations on page 47

CallEscapeBehavior

Purpose

Determines whether the driver calls a user-defined function or a stored procedure when JDBC Call escape syntax ({CALL PROC_NAME(...)} or {?=CALL FUNC_NAME(...)}) is used in a SQL statement.

Valid Values

select | call| callIfNoReturn

Behavior

If set to select, the driver calls a user-defined function.

If set to call, the driver calls a stored procedure.

If set to callIfNoReturn, the driver determines whether to call a user-defined function or a stored procedure based on whether a return value parameter (?=) is specified in the JDBC Call escape syntax. If a return value parameter is specified, the driver calls a user-defined function. If not, the driver calls a stored procedure.

Data Source Methods

```
public String getCallEscapeBehavior()
```

```
public void setCallEscapeBehavior(String)
```

Default

callIfNoReturn

Data Type

String

CatalogOptions

Purpose

Determines which type of metadata information is included in result sets when an application calls DatabaseMetaData methods.

Valid Values

2 | 4

Behavior

If set to 2, the driver queries database catalogs for column information.

If set to 4, a hint is provided to the driver to emulate getColumns() calls using the ResultSetMetaData object instead of querying database catalogs for column information. Using emulation can improve performance because the SQL statement that is formulated by the emulation is less complex than the SQL statement that is formulated using getColumns(). The argument to getColumns() must evaluate to a single table. If it does not, because of a wildcard or null value, for example, the driver reverts to the default behavior for getColumns() calls.

Data Source Methods

```
public Integer getCatalogOptions()
```

```
public void setCatalogOptions(Integer)
```

Default

2

Data Type

int

ClientHostName

Purpose

Specifies the host name of the client machine. This value is stored locally and is used for database administration/monitoring purposes.

Valid Values

string

where:

string

is the host name of the client machine.

Data Source Methods

```
public String getClientHostName()
```

```
public void setClientHostName(String)
```

Default

No default value

Data Type

String

See also

Using client information on page 47

ClientUser

Purpose

Specifies the user ID. This value is stored locally and is used for database administration/monitoring purposes.

Valid Values

string

where:

string

is a valid user ID.

Data Source Methods

`public String getClientUser()`

`public void setClientUser(String)`

Default

No default value

Data Type

String

See also

Using client information on page 47

ConnectionRetryCount

Purpose

Specifies the number of times the driver retries connection attempts to the primary database server, and if specified, alternate servers before returning a connection failure.

Valid Values

`0` | *x*

where:

x

is a positive integer that represents the number of retries.

Behavior

If set to `0`, the driver does not try to reconnect after the initial unsuccessful attempt.

If set to *x*, the driver retries connection attempts the specified number of times. If a connection is not established during the retry attempts, the driver returns an exception that is generated by the last database server to which it tried to connect.

Example

If this property is set to 2 and alternate servers are specified using the AlternateServers property, the driver retries the list of servers (primary and alternate) twice after the initial retry attempt.

Notes

- If an application sets a login timeout value (for example, using DataSource.loginTimeout or DriverManager.loginTimeout), and the login timeout expires, the driver ceases connection attempts.
- The ConnectionRetryDelay property specifies the wait interval, in seconds, to occur between retry attempts.
- The driver will not try to reconnect to a server which is not found.

Data Source Methods

```
public Integer getConnectionRetryCount()
```

```
public void setConnectionRetryCount(Integer)
```

Default

5

Data Type

int

See also

Configuring failover

ConnectionRetryDelay

Purpose

Specifies the number of seconds the driver waits between connection retry attempts when ConnectionRetryCount is set to a positive integer.

Valid Values

`0` | *x*

where:

x

is a number of seconds.

Behavior

If set to `0`, the driver does not delay between retries.

If set to *x*, the driver waits between connection retry attempts the specified number of seconds.

Example

If ConnectionRetryCount is set to 2, this property is set to 3, and alternate servers are specified using the AlternateServers property, the driver retries the list of servers (primary and alternate) twice after the initial retry attempt. The driver waits 3 seconds between retry attempts.

Data Source Methods

```
public Integer getConnectionRetryDelay()
```

```
public void setConnectionRetryDelay(Integer)
```

Default

1 (second)

Data Type

int

See also

Configuring failover

ConvertNull

Purpose

Controls how data conversions are handled for null values.

Valid Values

0 | 1

Behavior

If set to 0, the driver does not perform the data type check if the value of the column is null. This allows null values to be returned even though a conversion between the requested type and the column type is undefined.

If set to 1, the driver checks the data type that is requested against the data type of the table column that stores the data. If a conversion between the requested type and column type is not defined, the driver generates an "unsupported data conversion" exception regardless of whether the column value is null.

Data Source Methods

```
public Integer getConvertNull()
```

```
public void setConvertNull(Integer)
```

Default

1

Data Type

int

CryptoProtocolVersion

Purpose

Specifies a cryptographic protocol or comma-separated list of cryptographic protocols that can be used when TLS/SSL is enabled using the EncryptionMethod connection property.

Valid Values

cryptographic_protocol [[, *cryptographic_protocol*]...]

where:

cryptographic_protocol

is one of the following cryptographic protocols:

TLSv1.2 | TLSv1.1 | TLSv1 | SSLv3 | SSLv2

Caution: To avoid vulnerabilities associated with SSLv3 and SSLv2, good security practices recommend using TLSv1 or higher.

Example

If your server supports TLSv1.1 and TLSv1.2, you can specify acceptable cryptographic protocols with the following key-value pair:

```
CryptoProtocolVersion=TLSv1.1,TLSv1.2
```

Notes

- When multiple protocols are specified, the driver uses the highest version supported by the server. If none of the specified protocols are supported by the server, the connection fails and the driver returns an error.
- When no value has been specified for CryptoProtocolVersion, the cryptographic protocol used depends on the highest protocol version supported by the server and the highest protocol version supported by the JDK. Refer to the database management system documentation for information on which cryptographic protocols are supported.

Data Source Methods

```
public String getCryptoProtocolVersion()
```

```
public void setCryptoProtocolVersion(String)
```

Default

No default value

Data Type

String

See also

- EncryptionMethod on page 77
- Data Encryption on page 42

DatabaseName

Purpose

Specifies the name of the database to which you want to connect.

Valid Values

string

where:

`string`

is the name of a Denodo database.

Data Source Methods

```
public String getDatabaseName()
```

```
public void setDatabaseName(String)
```

Default

No default value

Data Type

String

EnableCancelTimeout

Purpose

Determines whether a cancel request that is sent by the driver as the result of a query timing out is subject to the same query timeout value as the statement it cancels.

Valid Values

`true` | `false`

If set to `true`, the cancel request times out using the same timeout value, in seconds, that is set for the statement it cancels. For example, if your application calls `Statement.setQueryTimeout(5)` on a statement and that statement is cancelled because its timeout value was exceeded, the driver sends a cancel request that also will time out if its execution exceeds 5 seconds. If the cancel request times out, because the server is down, for example, the driver throws an exception indicating that the cancel request was timed out and the connection is no longer valid.

If set to `false`, the cancel request does not time out.

Data Source Methods

```
public Boolean getEnableCancelTimeout()
public void setEnableCancelTimeout(Boolean)
```

Default

`false`

Data Type

Boolean

EnablePrepareThreshold

Purpose

Determines whether server-side prepared statements are created during the prepareStatement() API call or deferred to the execute() API call.

Valid Values

`true` | `false`

Behavior

If set to `true`, the driver defers server-side prepared statements to the execute() API call.

If set to `false`, the driver creates server-side prepared statements during the prepareStatement() API call.

Notes

- When this property is set to `true`, the use of server-side prepared statements is defined by the `PrepareThreshold` property value.
- When this property is set to `false`, the existing prepared statement is used during the execute() API call.

Data Source Methods

```
public Boolean getEnablePrepareThreshold()
public void setEnablePrepareThreshold(Boolean)
```

Default

`true`

Data Type

Boolean

See also

PrepareThreshold

EncryptionMethod

Purpose

Determines whether data is encrypted and decrypted when transmitted over the network between the driver and database server.

Valid Values

`noEncryption | SSL | requestSSL`

Behavior

If set to `noEncryption`, data is not encrypted or decrypted.

If set to `SSL`, data is encrypted using SSL. If the database server does not support SSL, the connection fails and the driver throws an exception.

If set to `requestSSL`, the login request and data is encrypted using SSL. If the database server does not support SSL, the driver establishes an unencrypted connection.

Notes

- When SSL is enabled, the following properties also apply:
  - CryptoProtocolVersion
  - HostNameInCertificate
  - KeyStore (for SSL client authentication)
  - KeyStorePassword (for SSL client authentication)
  - KeyPassword (for SSL client authentication)
  - TrustStore
  - TrustStorePassword
  - ValidateServerCertificate

Data Source Methods

```
public String getEncryptionMethod()
```

```
public void setEncryptionMethod(String)
```

Default

`noEncryption`

Data Type

String

See also

Data encryption

Performance considerations

ExtendedColumnMetadata

Purpose

Determines how the driver returns column metadata when retrieving results with ResultSetMetaData methods.

Valid Values

`true` | `false`

Behavior

If set to `true`, the driver makes an additional roundtrip to the database to retrieve actual values for column metadata. For example, the driver returns nullability as `IS_NULLABLE`, `NOT_NULLABLE`, or `NULLABILITY_UNKNOWN`, depending on the actual status of the column.

If set to `false`, the driver returns only the values for column metadata hardcoded in the driver. For example, the driver returns nullability as `NULLABILITY_UNKNOWN`.

Notes

Setting ExtendedColumnMetadata to enabled may diminish performance because an additional roundtrip to the database is required to retrieve actual values for the column metadata.

Data Source Methods

```
public Boolean getExtendedColumnMetadata()
```

```
public void setExtendedColumnMetadata(Boolean)
```

Default

`false`

Data Type

Boolean

HostNameInCertificate

Purpose

Specifies a host name for certificate validation when SSL encryption is enabled (EncryptionMethod=`SSL`) and validation is enabled (ValidateServerCertificate=`true`). This property is optional and provides additional security against man-in-the-middle (MITM) attacks by ensuring that the server the driver is connecting to is the server that was requested.

Valid Values

`host_name` | #SERVERNAME#

where:

host_name

is a valid host name.

Behavior

If *host_name* is specified, the driver compares the specified host name to the DNSName value of the SubjectAlternativeName in the certificate. If the certificate does not have a SubjectAlternativeName, the driver compares the host name with the Common Name (CN) part of the certificate. If the values do not match, the connection fails and the driver throws an exception.

If #SERVERNAME# is specified, the driver compares the server name that is specified in the connection URL or data source of the connection to the DNSName value of the SubjectAlternativeName in the certificate. If the certificate does not have a SubjectAlternativeName, the driver compares the host name to the CN part of the certificate's Subject name. If the values do not match, the connection fails and the driver throws an exception. If multiple CN parts are present, the driver validates the host name against each CN part. If any one validation succeeds, a connection is established.

Notes

- If SSL encryption or certificate validation is not enabled, this property is ignored.
- If SSL encryption and validation is enabled and this property is unspecified, the driver uses the server name specified in the connection URL or data source of the connection to validate the certificate.

Data Source Methods

```
public String getHostNameInCertificate()
```

```
public void setHostNameInCertificate(String)
```

Default

No default value

Data Type

String

ImportStatementPool

Purpose

Specifies the path and file name of the file to be used to load the contents of the statement pool. When this property is specified, statements are imported into the statement pool from the specified file.

If the driver cannot locate the specified file when establishing the connection, the connection fails and the driver throws an exception.

Valid Values

string

where:

string

is the path and file name of the file to be used to load the contents of the statement pool.

Data Source Methods

```
public String getImportStatementPool()
public void setImportStatementPool(String)
```

Default

No default value

Data Type

String

See also

- Statement pooling on page 50
- Performance considerations on page 47
- Refer to "Statement Pool Monitor" in the *Progress DataDirect for JDBC Drivers Reference* for further details.

InitializationString

Purpose

Specifies one or multiple SQL commands to be executed by the driver after it has established the connection to the database and has performed all initialization for the connection. If the execution of a SQL command fails, the connection attempt also fails and the driver throws an exception indicating which SQL command or commands failed.

Valid Values

string

where:

string

is one or multiple SQL commands.

Multiple commands must be separated by semicolons. In addition, if this property is specified in a connection URL, the entire value must be enclosed in parentheses when multiple commands are specified.

Example

```
jdbc:datadirect:denodo://server1:9996;DatabaseName=test;
InitializationString=(command1;command2)
```

Data Source Methods

```
public String getInitializationString()
```

```
public void setInitializationString(String)
```

Default

No default value

Data Type

String

InsensitiveResultSetBufferSize

Purpose

Determines the amount of memory used by the driver to cache insensitive result set data.

Valid Values

`-1 | 0 |` *x*

where:

x

is a positive integer that represents the size of the memory buffer.

Behavior

If set to `-1`, the driver caches insensitive result set data in memory. If the size of the result set exceeds available memory, an OutOfMemoryException is generated. With no need to write result set data to disk, the driver processes the data efficiently.

If set to `0`, the driver caches insensitive result set data in memory, up to a maximum of 2 GB. If the size of the result set data exceeds available memory, the driver pages the result set data to disk. Because result set data may be written to disk, the driver may have to reformat the data to write it correctly to disk.

If set to *x*, the driver caches insensitive result set data in memory and uses this value to set the size (in KB) of the memory buffer for caching insensitive result set data. If the size of the result set data exceeds available memory, the driver pages the result set data to disk. Because the result set data may be written to disk, the driver may have to reformat the data to write it correctly to disk. Specifying a buffer size that is a power of 2 results in efficient memory use.

Data Source Methods

```
public Integer getInsensitiveResultSetBufferSize()
```

```
public void setInsensitiveResultSetBufferSize(Integer)
```

Default

`2048`

Data Type

Integer

See also

Performance considerations

JavaDoubleToString

Purpose

Determines which algorithm the driver uses when converting a double or float value to a string value. By default, the driver uses its own internal conversion algorithm, which improves performance.

Valid Values

`true` | `false`

Behavior

If set to `true`, the driver uses the JVM algorithm when converting a double or float value to a string value. If your application cannot accept rounding differences and you are willing to sacrifice performance, set this value to `true` to use the JVM conversion algorithm.

If set to `false`, the driver uses its own internal algorithm when converting a double or float value to a string value. This value improves performance, but slight rounding differences within the allowable error of the double and float data types can occur when compared to the same conversion using the JVM algorithm.

Data Source Methods

```
public Boolean getJavaDoubleToString()
```

```
public void setJavaDoubleToString(Boolean)
```

Default

`false`

Data Type

Boolean

KeyPassword

Purpose

Specifies the password that is used to access the individual keys in the keystore file when SSL is enabled (`EncryptionMethod=SSL`) and SSL client authentication is enabled on the database server. This property is useful when individual keys in the keystore file have a different password than the keystore file.

Valid Values

string

where:

string

is a valid password.

Data Source Methods

```
public String getKeyPassword()
```

```
public void setKeyPassword(String)
```

Default

No default value

Data Type

String

KeyStore

Purpose

Specifies the directory of the keystore file to be used when SSL is enabled (EncryptionMethod=SSL) and SSL client authentication is enabled on the database server. The keystore file contains the certificates that the client sends to the server in response to the server's certificate request.

This value overrides the directory of the keystore file that is specified by the javax.net.ssl.keyStore Java system property. If this property is not specified, the keystore directory is specified by the javax.net.ssl.keyStore Java system property.

Valid Values

string

where:

string

is a valid directory of a keystore file.

Notes

- The keystore and truststore files can be the same file.

Data Source Methods

```
public String getKeyStore()
```

```
public void setKeyStore(String)
```

Default

No default value

Data Type

String

KeyStorePassword

Purpose

Specifies the password that is used to access the keystore file when SSL is enabled (EncryptionMethod=SSL) and SSL client authentication is enabled on the database server. The keystore file contains the certificates that the client sends to the server in response to the server's certificate request.

This value overrides the password of the keystore file that is specified by the javax.net.ssl.keyStorePassword Java system property. If this property is not specified, the keystore password is specified by the javax.net.ssl.keyStorePassword Java system property.

Notes

- The keystore and truststore files can be the same file.

Valid Values

string

where:

string

is a valid password.

Data Source Methods

```
public String getKeyStorePassword()
```

```
public void setKeyStorePassword(String)
```

Default

No default value

Data Type

String

LoadBalancing

Purpose

Determines whether the driver uses client load balancing in its attempts to connect to the database servers (primary and alternate). You can specify one or multiple alternate servers by setting the AlternateServers property.

Valid Values

`true` | `false`

Behavior

If set to `true`, the driver uses client load balancing and attempts to connect to the database servers (primary and alternate) in random order. The driver randomly selects from the list of primary and alternate servers which server to connect to first. If that connection fails, the driver again randomly selects from this list of servers until all servers in the list have been tried or a connection is successfully established.

If set to `false`, the driver does not use client load balancing and connects to each server based on their sequential order (primary server first, then, alternate servers in the order they are specified).

Data Source Methods

```
public Boolean getLoadBalancing()
```

```
public void setLoadBalancing(Boolean)
```

Default

`false`

Data Type

Boolean

See also

Failover

LoginTimeout

Purpose

Specifies the amount of time, in seconds, that the driver waits for a connection to be established before timing out the connection request.

Valid Values

`0` | *x*

where:

x

is a positive integer that represents a number of seconds.

Behavior

If set to `0`, the driver does not time out a connection request.

If set to *x*, the driver waits for the specified number of seconds before returning control to the application and throwing a timeout exception.

Data Source Methods

```
public Integer getLoginTimeout()
```

```
public void setLoginTimeout(Integer)
```

Default

`0`

Data Type

Integer

MaxLongVarcharSize

Purpose

Determines the maximum size of LONGVARCHAR columns when described within the result set metadata.

Valid Values

x

where:

x

is an integer greater than or equal to 1 and less than or equal to 1,073,741,823.

Data Source Methods

```
public Integer getMaxLongVarcharSize()
```

```
public void setMaxLongVarcharSize(Integer)
```

Default

1,073,741,823

Data Type

Integer

MaxNumericPrecision

Purpose

Determines the maximum precision of NUMERIC columns when described within the result set metadata.

Valid Values

x

where:

x

is an integer greater than or equal to 1 and less than or equal to 1000.

Data Source Methods

```
public Integer getMaxNumericPrecision()
public void setMaxNumericPrecision(Integer)
```

Default

1000

Data Type

Integer

MaxNumericScale

Purpose

Determines the maximum scale of NUMERIC columns when described within the result set metadata.

Valid Values

x

where:

x

is an integer greater than or equal to 0 and less than or equal to 998.

Data Source Methods

```
public Integer getMaxNumericScale()
public void setMaxNumericScale(Integer)
```

Default

998

Data Type

Integer

MaxPooledStatements

Purpose

Specifies the maximum number of prepared statements to be pooled for each connection and enables the driver's internal prepared statement pooling when set to an integer greater than zero (0). The driver's internal prepared statement pooling provides performance benefits when the driver is not running from within an application server or another application that provides its own statement pooling.

Valid Values

`0` | *x*

where:

x

is a positive integer that represents a number of prepared statements to be cached.

Behavior

If set to `0`, the driver's internal prepared statement pooling is not enabled.

If set to *x*, the driver's internal prepared statement pooling is enabled and the driver uses the specified value to cache up to that many prepared statements created by an application. If the value set for this property is greater than the number of prepared statements that are used by the application, all prepared statements that are created by the application are cached. Because CallableStatement is a sub-class of PreparedStatement, CallableStatements also are cached.

Notes

When you enable statement pooling, your applications can access the Statement Pool Monitor directly with DataDirect-specific methods. However, you can also enable the Statement Pool Monitor as a JMX MBean. To enable the Statement Pool Monitor as an MBean, statement pooling must be enabled with MaxPooledStatements and the Statement Pool Monitor MBean must be registered using the RegisterStatementPoolMonitorMBean connection property.

Example

If the value of this property is set to `20`, the driver caches the last 20 prepared statements that are created by the application.

Data Source Methods

```
public Integer getMaxPooledStatements()
```

```
public void setMaxPooledStatements(Integer)
```

Default

`0`

Data Type

Integer

See also

- Statement pooling on page 50
- Performance considerations on page 47
- RegisterStatementPoolMonitorMBean on page 96
- Refer to "Statement Pool Monitor" in the *Progress DataDirect for JDBC Drivers Reference* for further details.

MaxStatements

Purpose

An alias for the MaxPooledStatements property.

MaxVarcharSize

Purpose

Determines the maximum size of VARCHAR columns when described within the result set metadata.

Valid Values

x

where:

x

is an integer greater than or equal to `1` and less than or equal to `10485760`.

Notes

When string functions are used within the Select list of a Select statement, the driver describes the resulting string value with a size of 10,485,760. For instance, concatenating two columns that are each defined as VARCHAR(10) will result in a VARCHAR(10485760) rather than a VARCHAR(20). The unnecessarily long size can result in undesirable behavior in some JDBC applications.

Data Source Methods

```
public Integer getMaxVarcharSize()
```

```
public void setMaxVarcharSize(Integer)
```

Default

10,485,760

Data Type

Integer

Password

Purpose

Specifies a password that is used to connect to your database. A password is required if user ID/Password authentication is enabled on your database. Contact your system administrator to obtain your password.

Valid Values

string

where:

string

is a valid password. The password is case-sensitive.

Data Source Methods

```
public String getPassword()
```

```
public void setPassword(String)
```

Default

No default value

Data Type

String

PortNumber

Purpose

The TCP port of the primary database server that is listening for connections to the Denodo database.

This property is supported only for data source connections.

Note: The driver connects to the 9996 port and any customized port that is configured on the PostgreSQL wire protocol.

Valid Values

port

where:

port

is the port number.

Data Source Methods

```
public Integer getPortNumber()
public void setPortNumber(Integer)
```

Default

9996

Data Type

Integer

PrepareThreshold

Purpose

Specifies the number of PreparedStatement executions that must occur before the driver begins using server-side prepared statements.

Valid Values

0 | 1 | *x*

where:

x

is a positive integer that represents the number of PreparedStatement executions at which the driver begins using server-side prepared statements.

Behavior

If set to 0, the driver never uses server-side prepared statements.

If set to 1, the driver uses server-side prepared statements by default.

If set to *x*, the driver uses server-side prepared statements when the number of PreparedStatement executions reaches the specified value.

Example

If PrepareThreshold is set to 5, the driver starts using server-side prepared statements on the fifth execution of the same PreparedStatement object.

Note

- This property is applicable when the EnablePrepareThreshold property is set to true.

- The value `0` is recommended when using a load balancer with a server that cannot support server-side prepare operations.
- The value `1` is recommended when an application executes the same parameterized DML operation multiple times.
- The value *x* is recommended when the server-side prepare operation is used only for frequently executing queries.

Data Source Methods

```
public Integer getPrepareThreshold()
public void setPrepareThreshold(Integer)
```

Default

`0`

Data Type

int

See also

EnablePrepareThreshold

QueryTimeout

ProgramID

Purpose

The driver and version information on the client to be stored in the database. This value is stored locally and is used for database administration/monitoring purposes.

Valid Values

`string`

where:

`string`

is a value that identifies the product and version of the driver on the client.

Example

`DDJ04200`

Data Source Methods

```
public String getProgramID()
public void setProgramID(String)
```

Default

No default value

Data Type

String

See also

Using client information on page 47

ProxyHost

Description

Identifies a proxy server to use for the first connection.

Valid Values

server_name | *IP_address*

where:

server_name

is the name of the proxy server, which may be qualified with the domain name.

IP_address

is an IP address, specified in either IPv4 or IPv6 format, or a combination of the two.

Data Source Methods

```
public String getProxyHost()
public void setProxyHost(String)
```

Default

No default value

Data Type

String

ProxyPassword

Purpose

Specifies the password needed to connect to a proxy server for the first connection.

Valid Values

password

where:

password

is a valid password for that server. Contact your system administrator to obtain a valid password.

Data Source Methods

```
public String getProxyPassword()
```

```
public void setProxyPassword(String)
```

Default

No default value

Data Type

String

ProxyPort

Purpose

Specifies the port number where the proxy server is listening for HTTP or HTTPS requests for the first connection.

Valid Values

port

where:

port

is the port number on which the proxy server is listening. Contact your system administrator to obtain the correct port.

Data Source Methods

```
public Integer getProxyPort()
```

```
public void setProxyPort(Integer)
```

Default Value

`0` which means that the default value is determined by whether the value specified for the ProxyHost property is an HTTP or HTTPS URL.

For HTTP: `80`

For HTTPS: `443`

Data Type

Integer

ProxyUser

Purpose

Specifies the user name needed to connect to a proxy server for the first connection.

Valid Values

user_name

where:

user_name

is a valid user ID for the proxy server.

Data Source Methods

```
public String getProxyUser()
```

```
public void setProxyUser(String)
```

Default Value

No default value

Data Type

String

QueryTimeout

Purpose

Sets the default query timeout (in seconds) for all statements created by a connection.

Valid Values

`-1` | `0` | *x*

where:

x

is a positive integer that represents a number of seconds.

Behavior

If set to `-1`, the query timeout functionality is disabled. The driver silently ignores calls to the Statement.setQueryTimeout() method.

If set to `0`, the default query timeout is infinite (the query does not time out).

If set to *x*, the driver uses the value as the default timeout for any statement that is created by the connection. To override the default timeout value set by this connection option, call the Statement.setQueryTimeout() method to set a timeout value for a particular statement.

Data Source Methods

```
public Integer getQueryTimeout()
```

```
public void setQueryTimeout(Integer)
```

Default

`0`

Data Type

Integer

RegisterStatementPoolMonitorMBean

Purpose

Registers the Statement Pool Monitor as a JMX MBean when statement pooling has been enabled with MaxPooledStatements. This allows you to manage statement pooling with standard JMX API calls and to use JMX-compliant tools, such as JConsole.

Valid Values

`true` | `false`

Behavior

If set to `true`, the driver registers an MBean for the statement pool monitor for each statement pool. This gives applications access to the Statement Pool Monitor through JMX when statement pooling is enabled.

If set to `false`, the driver does not register an MBean for the statement pool monitor for any statement pool.

Notes

Registering the MBean exports a reference to the Statement Pool Monitor. The exported reference can prevent garbage collection on connections if the connections are not properly closed. When garbage collection does not take place on these connections, out of memory errors can occur.

Data Source Methods

```
public Boolean getRegisterStatementPoolMonitorMBean()
```

```
public void setRegisterStatementPoolMonitorMBean(Boolean)
```

Default

`false`

Data Type

Boolean

See also

- MaxPooledStatements on page 88
- Refer to "Statement Pool Monitor" in the *Progress DataDirect for JDBC Drivers Reference* for further details.

ResultSetMetaDataOptions

Purpose

Determines whether the driver returns table name information in the ResultSet metadata for Select statements.

Valid Values

`0 | 1`

Behavior

If set to `0` and the ResultSetMetaData.getTableName() method is called, the driver does not perform additional processing to determine the correct table name for each column in the result set. The getTableName() method may return an empty string for each column in the result set.

If set to `1` and the ResultSetMetaData.getTableName() method is called, the driver performs additional processing to determine the correct table name for each column in the result set. The driver returns schema name and catalog name information when the ResultSetMetaData.getSchemaName() and ResultSetMetaData.getCatalogName() methods are called if the driver can determine that information.

Data Source Methods

```
public Integer getResultSetMetaDataOptions()
```

```
public void setResultSetMetaDataOptions(Integer)
```

Default

`0`

Data Type

Integer

See also

Performance considerations

ServerName

Purpose

REQUIRED. Specifies either the IP address in IPv4 or IPv6 format, or the server name (if your network supports named servers) of the primary database server.

This property is supported only for data source connections.

Valid Values

string

where:

string

is a valid IP address or server name.

Example

`122.23.15.12` or `DenodoServer`

Data Source Methods

```
public String getServerName()
```

```
public void setServerName(String)
```

Default

No default value

Data Type

String

SpyAttributes

Purpose

Enables DataDirect Spy to log detailed information about calls issued by the driver on behalf of the application. DataDirect Spy is not enabled by default.

Valid Values

(*spy_attribute* [; *spy_attribute*]...)

where:

spy_attribute

is any valid DataDirect Spy attribute.

Refer to "Tracking JDBC Calls with DataDirect Spy" in the *Progress DataDirect for JDBC Drivers Reference* for a list of supported attributes.

Example

The following value instructs the driver to log all JDBC activity to a file using a maximum of 80 characters for each line.

```
(log=(file)/tmp/spy.log;linelimit=80)
```

Notes

- If coding a path on Windows to the log file in a Java string, the backslash character (\) must be preceded by the Java escape character, a backslash. For example: `log=(file)C:\\temp\\spy.log`.

Data Source Methods

```
public String getSpyAttributes()
```

```
public void setSpyAttributes(String)
```

Default

No default value

Data Type

String

SupportsCatalogs

Purpose

Enables support for catalogs. While Denodo has the notion of catalogs (or databases), you can only select from tables that reside within the catalog you specified at connect time. Catalogs cannot be changed after connecting. Therefore, most applications behave better if you do not indicate support for catalogs.

Valid Values

`true` | `false`

Behavior

If set to `true`, the driver returns the database as the catalog for catalog calls, for example, getTables and getColumns.

If set to `false`, the driver returns NULL for the catalog in catalog calls.

Notes

The SupportsCatalogs connection property affects the following catalog methods:

- getCatalogSeparator
- getCatalogTerm
- getMaxCatalogNameLength

- isCatalogAtStart
- supportsCatalogsInDataManipulation
- supportsCatalogsInProcedureCalls
- supportsCatalogsInTableDefinitions
- supportsCatalogsInIndexDefinitions
- supportsCatalogsInPrivilegeDefinitions

Data Source Methods

```
public Boolean getSupportsCatalogs()
```

```
public void setSupportsCatalogs(Boolean)
```

Default

`true`

Data Type

Boolean

TransactionErrorBehavior

Purpose

Determines how the driver handles errors that occur within a transaction. When an error occurs in a transaction, the Denodo server does not allow any operations on the connection except for rolling back the transaction.

Valid Values

```
none|RollbackTransaction|RollbackSavepoint
```

Behavior

If set to `none`, the driver does not roll back the transaction when an error occurs. The application must handle the error and roll back the transaction. Any operation on the statement other than a rollback results in an error.

If set to `RollbackTransaction`, the driver rolls back the transaction when an error occurs. In addition to the original error message, the driver posts an error message indicating that the transaction has been rolled back.

If set to RollbackSavepoint, the driver rolls back the transaction to the last savepoint when an error is detected. In manual commit mode, the driver automatically sets a savepoint after each statement issued. This value makes transaction behavior resemble that of most other database system types, but uses more resources on the database server and may incur a slight performance penalty.

Data Source Methods

```
public String getTransactionErrorBehavior()
```

```
public void setTransactionErrorBehavior(String)
```

Default

`RollbackTransaction`

Data Type

String

TrustStore

Purpose

Specifies the directory of the truststore file to be used when SSL is enabled (EncryptionMethod=SSL) and server authentication is used. The truststore file contains a list of the Certificate Authorities (CAs) that the client trusts.

This value overrides the directory of the truststore file that is specified by the javax.net.ssl.trustStore Java system property. If this property is not specified, the truststore directory is specified by the javax.net.ssl.trustStore Java system property.

This property is ignored if ValidateServerCertificate=`false`.

Valid Values

`string`

The directory of the truststore file.

Data Source Methods

```
public String getTrustStore()
```

```
public void setTrustStore(String)
```

Default

No default value

Data Type

String

TrustStorePassword

Purpose

Specifies the password that is used to access the truststore file when SSL is enabled (EncryptionMethod=SSL) and server authentication is used. The truststore file contains a list of the Certificate Authorities (CAs) that the client trusts.

This value overrides the password of the truststore file that is specified by the javax.net.ssl.trustStorePassword Java system property. If this property is not specified, the truststore password is specified by the javax.net.ssl.trustStorePassword Java system property.

This property is ignored if ValidateServerCertificate=`false`.

Valid Values

`string`

where:

`string`

is a valid password for the truststore file.

Data Source Methods

```
public String getTrustStorePassword()
```

```
public void setTrustStorePassword(String)
```

Default

No default value

Data Type

String

User

Purpose

Specifies the user name that is used to connect to the database. A user name is required if user ID/password authentication is enabled on your database. Contact your system administrator to obtain your user name.

Valid Values

`string`

where:

`string`

is a valid user name. The user name is case-sensitive.

Data Source Methods

```
public String getUser()
```

```
public void setUser(String)
```

Default

No default value

Data Type

String

ValidateServerCertificate

Purpose

Determines whether the driver validates the certificate that is sent by the database server when SSL encryption is enabled (EncryptionMethod=SSL). When using SSL server authentication, any certificate that is sent by the server must be issued by a trusted Certificate Authority (CA). Allowing the driver to trust any certificate that is returned from the server even if the issuer is not a trusted CA is useful in test environments because it eliminates the need to specify truststore information on each client in the test environment.

Valid Values

`true` | `false`

Behavior

If set to `true`, the driver validates the certificate that is sent by the database server. Any certificate from the server must be issued by a trusted CA in the truststore file. If the HostNameInCertificate property is specified, the driver also validates the certificate using a host name. The HostNameInCertificate property is optional and provides additional security against man-in-the-middle (MITM) attacks by ensuring that the server the driver is connecting to is the server that was requested.

If set to `false`, the driver does not validate the certificate that is sent by the database server. The driver ignores any truststore information that is specified by the TrustStore and TrustStorePassword properties or Java system properties.

Notes

- Truststore information is specified using the TrustStore and TrustStorePassword properties or by using Java system properties.

Data Source Methods

```
public Boolean getValidateServerCertificate()
```

```
public void setValidateServerCertificate(Boolean)
```

Default

`true`

Data Type

Boolean

VarcharClobThreshold

Purpose

Enables CLOB functionality when handling character data. Determines whether columns of the *Character varying* data type will be described as VARCHAR or LONGVARCHAR (CLOB).

Valid Values

x

where:

x

is a number of characters.

Behavior

If the width of a *Character varying* column is greater than *x*, the *Character varying* data type is described as LONGVARCHAR and CLOB functionality is enabled. The driver allows you to retrieve and update long data by using JDBC methods designed for Clobs. This functionality provides random access to data and allows searching for patterns in the data. To provide these benefits, data must be cached. Because data is cached, your application will incur a performance penalty, particularly if data is read once sequentially.

If the width of a *Character varying* column is less than or eqal to *x*, the *Character varying* data type is described as VARCHAR. If you want to avoid the performance penalties associated with CLOB functionality, you should set this value at a value greater than the maximum *Character varying* column width your application handles.

Data Source Methods

```
public Varchar getVarcharClobThreshold()
```

```
public void setVarcharClobThreshold(Varchar)
```

Default

32768

Data Type

VARCHAR or LONGVARCHAR

See also

Performance considerations on page 47

Large object (LOB) support on page 53