Progress DataDirect for JDBC for Amazon Redshift

User's Guide

Release 6.0.0

Copyright

Updated: 2025/10/23

Table of Contents

1

Welcome to the Progress DataDirect for JDBC for Amazon Redshift Driver

The Progress® DataDirect® for JDBC™ for Amazon Redshift™ driver supports standard SQL query language to select, insert, update, and delete data from Amazon Redshift data warehouses. The driver provides access to data in Redshift that can be pulled into a data visualization tool to get important insights into engineering operations. This functionality allows seamless integration with third-party software and provides the most comprehensive SQL support and JDBC standard connectivity to BI (Business Intelligence) and ETL (Extract,Transform, Load) tools.

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-amazon-redshift.

For details, see the following topics:

What's new in this release?

Changes Since 6.0.0 GA

- **Driver enhancements**
  - The driver has been enhanced to comply with FIPS standards for data encryption. As part of this enhancement, the driver was tested with FIPS 140-3 enabled using a Red Hat OpenJDK 21 on a Red Hat Universal Base Image 9 instance. See FIPS (Federal Information Processing Standard) on page 46 for details.
- **Changed Behavior**
  - The connection property `SpyAttributes` has been updated to exclude the attribute `load=classname`, which was previously used to load the driver specified by the given class name. See SpyAttributes on page 96 for details.

Changes for 6.0.0 Release

- The driver provides proxy support. See Proxy server on page 14 and Connection property descriptions on page 59 for more information.
- The driver has been enhanced to support the following data types:
  - Text
  - Time
  - Time with time zone
  - Timestamp with time zone

  Note: For more information, see Data types on page 15.
- The driver has been enhanced to include timestamp in the Spy and JDBC packet logs by default. If required, you can disable the timestamp logging by specifying the following at connection: For Spy logs, set spyAttributes=(log=(file)Spy.log;timestamp=no) and for JDBC packet logs, set ddtdbg.ProtocolTraceShowTime=false.
- The driver has been enhanced to support stored procedures for Redshift.
- The driver supports the CallEscapeBehavior connection property. It determines whether the driver calls a user-defined function or a stored procedure when JDBC Call escape syntax is used in a SQL statement. See CallEscapeBehavior on page 69 for details.
- Interactive SQL for JDBC (JDBCISQL) is now installed with the product. JDBCISQL is a command-line interface that supports connecting your driver to a data source, executing SQL statements and retrieving

results in a terminal. This tool provides a method to quickly test your drivers in an environment that does not support GUIs. See Interactive SQL for JDBC (JDBCISQL) on page 28 for details.

- The driver no longer registers the Statement Pool Monitor as a JMX MBean by default. To register the Statement Pool Monitor and manage statement pooling with standard JMX API calls, the new RegisterStatementPoolMonitorMBean connection property must be set to true.

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\redshift.jar
     ```
   - **UNIX/LINUX Example**

     ```
     CLASSPATH=.:/opt/Progress/DataDirect/JDBC/lib/60/redshift.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 show how to connect using user ID and password authentication.

UserID/Password

```
jdbc:datadirect:redshift://MyServer:5439;DatabaseName=MyDB;
User=JSmith;Password=secret;
```

Note: See User ID/password authentication on page 13 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.
   - Supported SQL statements and extensions: This section describes the syntax used for SQL statements supported by the driver. You can modify and use the provided examples for your application or tool.

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.redshift.RedshiftDriver

DataSource class:

com.ddtek.jdbcx.redshift.RedshiftDataSource

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:redshift://server_name:port_number;DatabaseName=database_name;
User=user_name;Password=password;[property=value[;...]];
```

where:

server_name

specifies either the IP address or the server name of the primary database server.

port_number

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

database_name

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

user_name

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

password

specifies a password that is used to connect to your Redshift 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:redshift://MyServer:5439;DatabaseName=MyDB;
  User=JSmith;Password=secret;");
```

Proxy server

This string includes the properties you may need to connect through a proxy server with UserID/Password authentication.

```
jdbc:datadirect:redshift://server_name:port_number;
DatabaseName=database_name;ProxyHost=proxy_host;ProxyPassword=proxy_password;
ProxyPort=proxy_port;ProxyUser=proxy_user;User=user_name;Password=password;
[property=value[;...]];
```

where:

server_name

specifies either the IP address or the server name of the primary database server.

port_number

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

database_name

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

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 ID that is used to connect to the Redshift instance. For example, `jsmith@example.com`.

password

specifies the password used to connect to your Redshift instance.

property=value

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

The following example connection string includes the properties required for using a proxy server with UserID/Password authentication.

```
Connection conn = DriverManager.getConnection
("jdbc:datadirect:redshift://MyServer:5439;DatabaseName=MyDB;
  ProxyHost=pserver;ProxyPassword=proxysecret;ProxyPort=808;ProxyUser=PUser;
  User=jsmith;Password=secret;");
```

See also

Connection property descriptions on page 59

Data types

The following table lists the data types supported by the driver and describes how they are mapped to JDBC data types. See "getTypeInfo()" for getTypeInfo() results of data types supported by the driver.

Table 1: Amazon Redshift Data Types

Redshift Data Type	JDBC Data Type
Bigint	BIGINT
Boolean	BOOLEAN
Character	CHAR
Character varying[1]	VARCHAR
Date	DATE
Double precision	DOUBLE
Integer	INTEGER
Numeric	NUMERIC
Real	REAL
Smallint	SMALLINT
Text	LONGVARCHAR
Time	TIMESTAMP
Time with time zone	TIMESTAMP
Timestamp	TIMESTAMP
Timestamp with time zone	TIMESTAMP

[1] You can determine how these columns are described by setting the VarcharClobThreshold connection property.

See also

getTypeInfo() on page 16
VarcharClobThreshold on page 103

getTypeInfo()

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

Table 2: getTypeInfo()

TYPE_NAME = bigint 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 = Bigint 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 = boolean AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 16 (BOOLEAN) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' 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 = NULL

TYPE_NAME = character AUTO_INCREMENT = NULL CASE_SENSITIVE = true CREATE_PARAMS = *length* DATA_TYPE = 1 (CHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = Character MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 4096 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = character varying [2] AUTO_INCREMENT = NULL CASE_SENSITIVE = true CREATE_PARAMS = *max length* DATA_TYPE = 12 (VARCHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = Character varying MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 65535 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = date AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 91 (DATE) FIXED_PREC_SCALE = false LITERAL_PREFIX = {d' 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

[2] You can determine how these columns are described by setting the VarcharClobThreshold on page 103 connection property.

TYPE_NAME = double precision 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 precision MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = 2 PRECISION = 53 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false
TYPE_NAME = integer 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 = Integer 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 = numeric AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = *precision, scale* DATA_TYPE = 2 (NUMERIC) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = Numeric MAXIMUM_SCALE = 37	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 38 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false

TYPE_NAME = real AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = *precision* DATA_TYPE = 7 (REAL) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = Real MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = 2 PRECISION = 24 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false
TYPE_NAME = smallint AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 5 (SMALLINT) FIXED_PREC_SCALE = false LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = Smallint MAXIMUM_SCALE = 0	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 5 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 = -1 (LONGVARCHAR) 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 = 256 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

TYPE_NAME = time AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = *fractional_seconds_precision* DATA_TYPE = 93 (TIME) FIXED_PREC_SCALE = false LITERAL_PREFIX = {t' LITERAL_SUFFIX = '} 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 = NULL
TYPE_NAME = time with time zone AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = *fractional_seconds_precision* DATA_TYPE = 93 (TIMESTAMP) FIXED_PREC_SCALE = false LITERAL_PREFIX = {t' LITERAL_SUFFIX = '} LOCAL_TYPE_NAME = Time with time zone 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 = NULL

TYPE_NAME = timestamp AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = *fractional_seconds_precision* DATA_TYPE = 93 (TIMESTAMP) FIXED_PREC_SCALE = false LITERAL_PREFIX = {ts' LITERAL_SUFFIX = '} 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 = NULL
TYPE_NAME = timestamp with time zone AUTO_INCREMENT = NULL CASE_SENSITIVE = false CREATE_PARAMS = *fractional_seconds_precision* DATA_TYPE = 93 (TIMESTAMP) FIXED_PREC_SCALE = false LITERAL_PREFIX = {ts' LITERAL_SUFFIX = '} LOCAL_TYPE_NAME = Timestamp with time zone MAXIMUM_SCALE = 6	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 34 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

Driver specifications

This section describes the general functionality supported by the driver.

- **Unicode support**: Multilingual JDBC applications can be developed on any operating system using the driver to access both Unicode and non-Unicode enabled databases. Internally, Java applications use UTF-16 Unicode encoding for string data. When fetching data, the driver automatically performs the conversion from the character encoding used by the database to UTF-16. Similarly, when inserting or updating data in the database, the driver automatically converts UTF-16 encoding to the character encoding used by the database.

  The JDBC API provides mechanisms for retrieving and storing character data encoded as Unicode (UTF-16) or ASCII. Additionally, the Java String object contains methods for converting UTF-16 encoding of string data to or from many popular character encodings.

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

- **Scrollable cursor support**: The driver supports scroll-insensitive result sets and updatable result sets.

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.

- **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.

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:

   ```
   redshift.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 examples show how to connect using user ID and password authentication.

     ```
     jdbc:datadirect:redshift://MyServer:5439;DatabaseName=MyDB;
     User=JSmith;Password=secret;
     ```

     Note: See User ID/password authentication on page 13 for details.

   - **Dialect**: Select **SQL92** (the default) from the drop-down box.
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\redshift.jar
   ```

   Linux

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

5. Provide values for the following fields; then, close the Driver Manager window.

- **Name**: Type an alias for your driver. For example:

  ```
  Redshift
  ```

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

  ```
  jdbc:datadirect:redshift://MyServer:5439
  ```

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

  ```
  com.ddtek.jdbc.redshift.RedshiftDriver
  ```

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 examples show how to connect using user ID and password authentication.

     ```
     jdbc:datadirect:redshift://MyServer:5439;DatabaseName=MyDB;
     User=JSmith;Password=secret;
     ```

   Note: See Authentication 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 `TESTNEWTIME` table:

   ```
   SELECT * FROM TESTNEWTIME
   ```

To select the URLs for a specified issue:

```
SELECT * FROM TESTNEWTIMETZ WHERE TIMETZCOLUMN = <ID>
```

See "Supported SQL statements and extensions" for the supported syntax used to execute SQL statements.

Note: If you are fetching large sets of data, you may want to limit the results using the Max Rows and Max Chars fields.

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.redshift.RedshiftDriver
   ```

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

   ```
   connect jdbc:datadirect:redshift://MyServer:5439;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 INFORMATION_SCHEMA.SYSTEM_TABLES;
   ```

   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/redshift.jar
```

Windows Example

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

UNIX Example

```
CLASSPATH=.:/opt/Progress/DataDirect/JDBC/lib/60/redshift.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 examples show, this method specifies a string containing a connection URL.

UserID/Password authentication

```
Connection conn = DriverManager.getConnection
("jdbc:datadirect:redshift://MyServer:5439;DatabaseName=MyDB;
  User=JSmith;Password=secret;");
```

Note: See User ID/password authentication on page 13 for details.

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 UserID/Password authentication.

Connection URL Syntax

The connection URL takes the following form:

```
jdbc:datadirect:redshift://server_name:port_number;DatabaseName=database_name;
UserName=user_name;Password=password;[property=value[;...]];
```

where:

server_name

specifies either the IP address or the server name of the primary database server.

port_number

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

database_name

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

user_name

Specifies the user ID used to authenticate to Redshift with UserID/Password authentication method.

password

Specifies the password used to authenticate to Redshift with UserID/Password authentication method.

Important: The password is a confidential value used to authenticate to the server. To prevent unauthorized access, this value must be securely maintained.

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 UserID/Password authentication.

```
Connection conn = DriverManager.getConnection
("jdbc:datadirect:redshift://MyServer:5439;DatabaseName=MyDB;
  UserName=JSmith;Password=secret");
```

See also

Connection property descriptions on page 59
Connection URL examples on page 13

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:

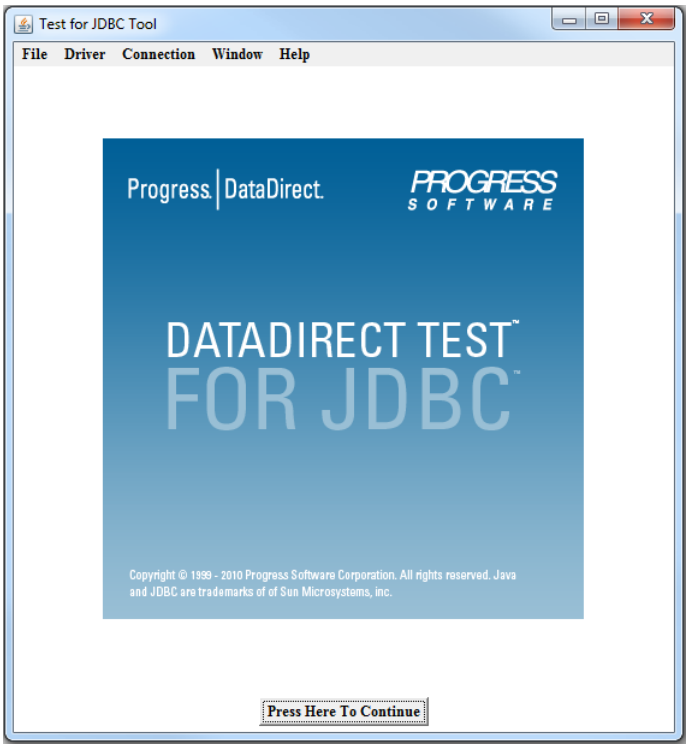
1. Navigate to the installation directory. The default location is:
   - Windows systems: `Program Files\Progress\DataDirect\JDBC\testforjdbc`
   - UNIX and Linux systems: `/opt/Progress/DataDirect/JDBC/testforjdbc`

   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:

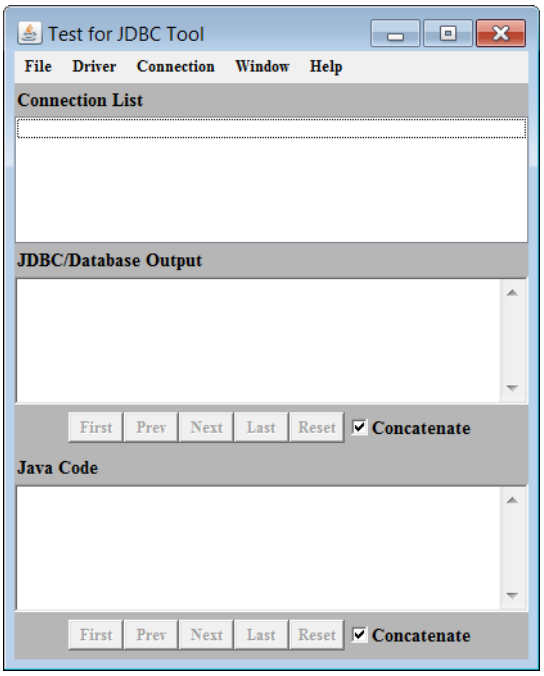
- `testforjdbc.bat` (on Windows systems)
- `testforjdbc.sh` (on UNIX and Linux systems)

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



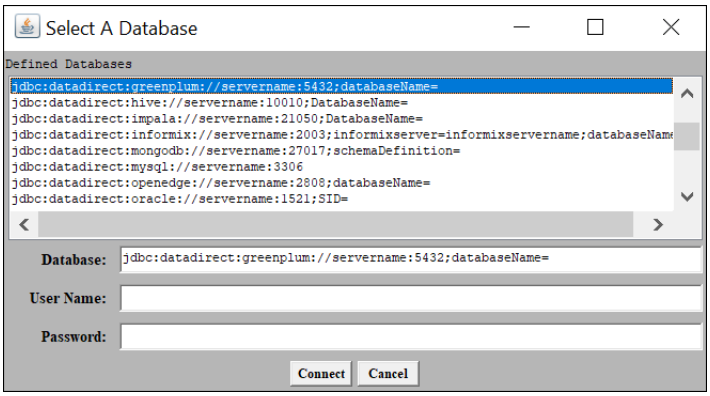
3. Click **Press Here to Continue**.

   The main dialog appears:



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

   The **Select A Database** dialog appears:



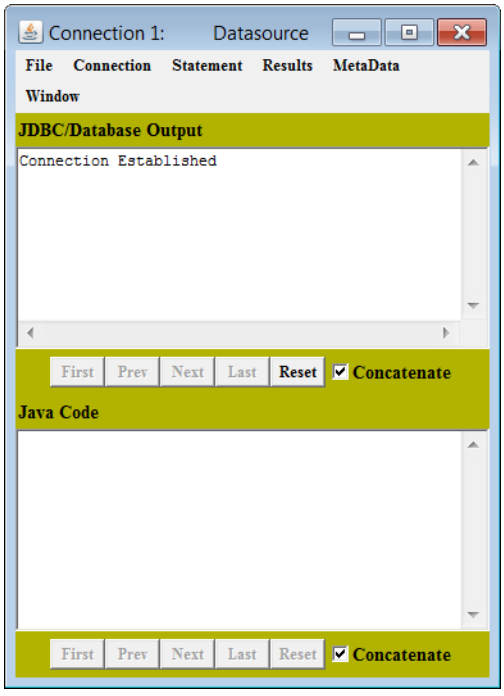
5. Select the appropriate database template from the **Defined Databases** field.
6. In the **Database** field, specify all required connection properties.

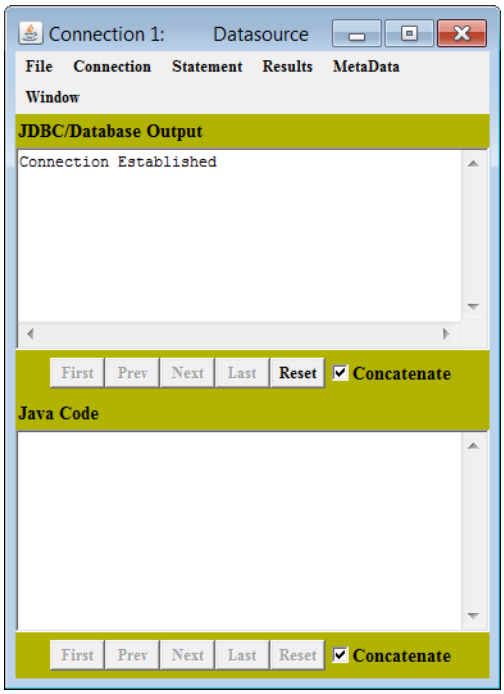
   For example:

   ```
   jdbc:datadirect:redshift://server3:5439;
   ```

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 `DataSource` 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.redshift.RedshiftDataSource;
```

Next, you will need to set the values and define the data source. For example, the following definition contains the minimum properties required for a connection using the UserID/Password authentication.

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").

```
RedshiftDataSource mds = new RedshiftDataSource();
mds.setDescription("My Redshift Data Source");
mds.setServerName("MyServer");
mds.setPortNumber(5439);
mds.setUserName("JSmith");
mds.setPassword ("Password");
```

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 instanceof RedshiftDataSource)
{
RedshiftDataSource jmds = (RedshiftDataSource) ds;
System.out.println("description=" + jmds.getDescription());
System.out.println("serverName=" + jmds.getServerName());
System.out.println("portNumber=" + jmds.getPortNumber());
System.out.println("userName=" + jmds.getUserName());
System.out.println("password=" + jmds.getPassword());
...
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.

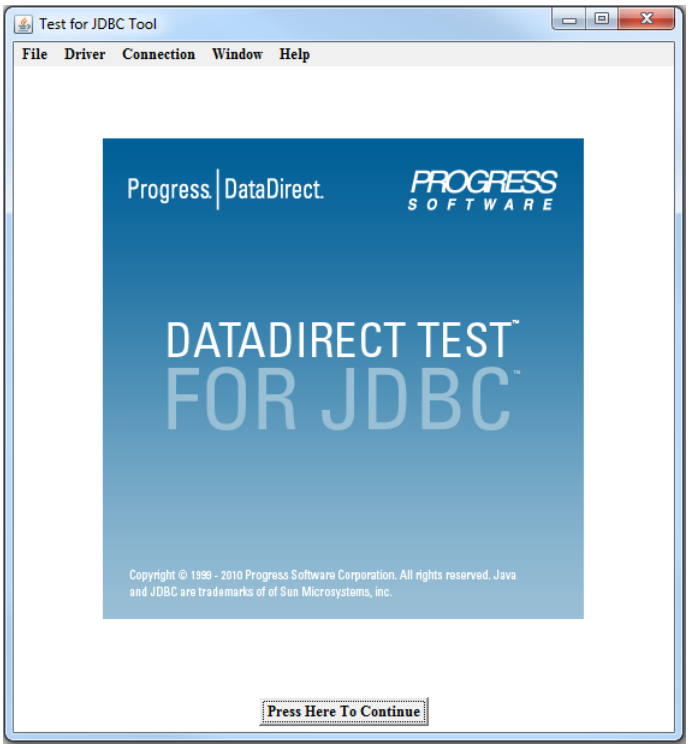
Take the following steps to establish a connection.

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.

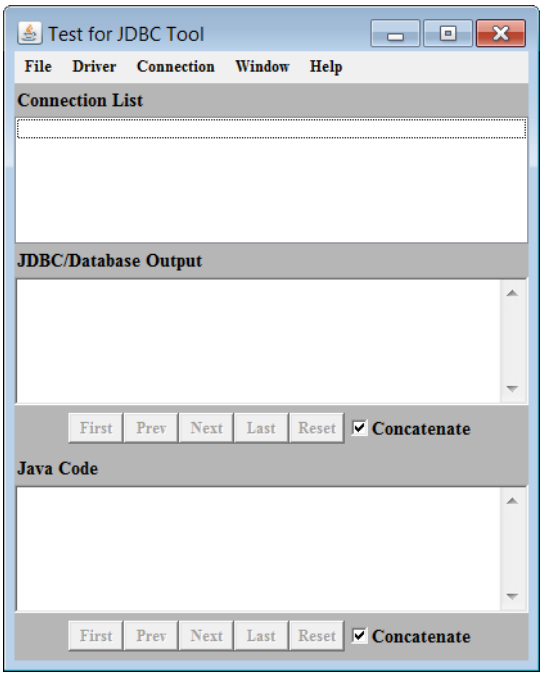
2. From the `testforjdbc` folder, run the platform-specific tool:
   - `testforjdbc.bat` (on Windows systems)
   - `testforjdbc.sh` (on UNIX and Linux systems)

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



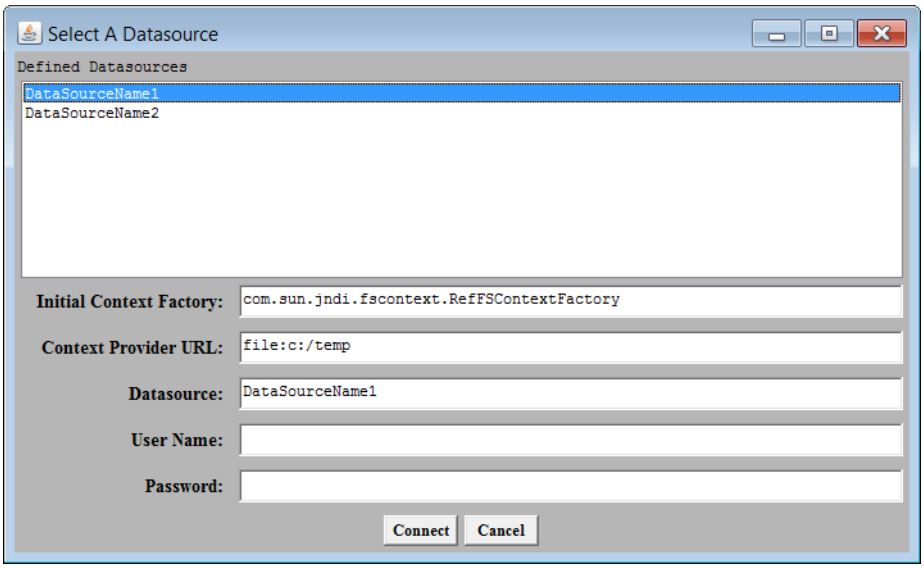
3. Click **Press Here to Continue**.

   The main dialog appears:



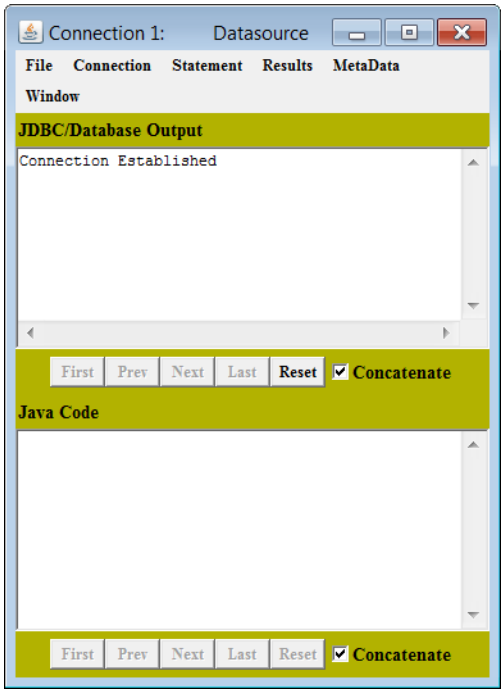
4. From the menu bar, select **Connection > Connect to DB via Data Source**.

   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 *User ID/password authentication*. It authenticates the user to the database using a user name and password.

Take the following steps to configure user ID/Password authentication.

1. Set the ServerName property to the IP address or server name of the primary database server.
2. Set the PortName property to the TCP port of the primary database server that is listening for connections to the database.
3. Set the DatabaseName property to the name of the database to which you want to connect.
4. Set the User property to provide the user ID.
5. Set the Password property to provide the password.

For example, the following is a connection string with only the required properties for making a connection using user ID/password authentication.

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

```
Connection conn = DriverManager.getConnection
("jdbc:datadirect:redshift://MyServer:5439;DatabaseName=MyDB;
  User=JSmith;Password=secret);
```

See also

Password on page 89
User on page 102

Performance considerations

BatchMechanism: If your application does not require individual update counts for each statement or parameter set in the batch, then BatchMechanism should be set to MultiRowInsert. Unlike the native batch mechanism, the multi-row insert mechanism only returns the total number of update counts for batch inserts. Setting BatchMechanism to MultiRowInsert therefore offers substantial performance gains when performing batch inserts.

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.

CatalogOptions: To improve performance, the driver can emulate getColumns() calls using the ResultSetMetaData object instead of querying database catalogs for the column information. Using emulation can improve performance because the SQL statement formulated by the emulation is less complex than the SQL statement 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.

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.

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.

See also

Connection property descriptions on page 59

Failover support

The driver provides connection failover support to ensure continuous, uninterrupted access to data. *Connection failover* provides failover protection for new connections. In more traditional scenarios, the driver fails over new connections to an alternate, or backup, database server if the primary database server is unavailable. However, Amazon Redshift currently supports only the notion of a single leader node. Therefore, to ensure a continuous connection, you only need to set the ConnectionRetryCount and ConnectionRetryDelay connection properties.

See also

ConnectionRetryCount on page 72
ConnectionRetryDelay on page 73

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 Amazon Redshift datasource.

Note: Connection hangs can occur when the driver is configured for SSL and the database server does not support 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 SSL.

To configure SSL encryption:

Important: The driver complies with FIPS when FIPS mode is enabled with the client JVM. See "FIPS (Federal Information Processing Standard)" for more information.

- Set the ServerName property to the name or the IP address of the Redshift 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 `5439`.
- Set the EncryptionMethod property to `SSL`.
- (Optional) Set the CryptoProtocolVersion property to specify acceptable cryptographic protocol versions (for example, TLSv1.2) supported by your server.
- (Optional) 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) 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.

The following examples demonstrate the required properties for a session using TLS/SSL encryption with no authentication.

For a connection URL:

```
Connection conn = DriverManager.getConnection
 ("jdbc:datadirect:redshift://myserver:5439;DatabaseName=mydb;
   EncryptionMethod=SSL");
```

For a data source:

```
RedshiftDataSource mds = new RedshiftDataSource();
mds.setDescription("My Redshift Data Source");
mds.setDatabaseName("mydb");
mds.setEncryptionMethod("SSL");
mds.setServerName("myserver");
```

See also

Connection property descriptions on page 59
Configuring TLS/SSL Server Authentication on page 44
Configuring TLS/SSL Client Authentication on page 45

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.

FIPS (Federal Information Processing Standard)

The Federal Information Processing Standard (or FIPS) is a cryptography standard created by the U.S. government. FIPS specifications require certain secure algorithms, cryptographic modules, and random number generation. The driver is FIPS compliant for data encryption when FIPS is enabled for the JVM on the client machine.

The following applies when the driver is running in a FIPS environment:

- The driver complies with 140-3 and 140-2 standards.
- The driver uses PKCS #11 providers to access keystores.

The driver was tested with FIPS 140-3 enabled using Red Hat OpenJDK 21 on a Red Hat Universal Base Image 9 instance.

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:

- Parameter metadata support
- ResultSet metadata support
- Client information
- Large object (LOB) support
- Batch inserts and updates
- DataDirect Bulk Load
- Using CSV files

Parameter metadata support

The driver supports returning parameter metadata.

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.

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.

Large object (LOB) support

The driver allows you to retrieve and update long data, specifically LONGVARCHAR[3] data, using JDBC methods designed for Clobs. When using these methods to update long data as Clobs, the updates are made to the local copy of the data contained in the Clob object.

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

Batch inserts and updates

To execute batch operations, the driver can use either of the following batch mechanisms:

- The Amazon Redshift data warehouse native batch mechanism. The native batch mechanism returns individual update counts for each statement or parameter set in the batch as required by the JDBC 3.0 specification.
- A mechanism internal to the driver that uses a parameterized multi-row insert statement. The multi-row insert batch mechanism returns only the total number of update counts in the batch, but provides substantial performance gains when performing batch inserts. The multi-row insert batch mechanism only applies to Insert statements for batch executes called with PreparedStatement objects. If a command other than the Insert statement is used, the driver uses the native batch mechanism to execute the command. The multi-row insert batch mechanism is the default.

See also

BatchMechanism on page 68

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

DataDirect Bulk Load

The driver emulates bulk load using the standard batch mechanism because Amazon Redshift does not have native bulk load support. You can perform bulk load operations by creating a DDBulkLoad object and using the methods provided by the DDBulkLoad interface in the com.ddtek.jdbc.extensions package for bulk load. See "Using a DDBulkLoad Object" and "BulkLoadBatchSize."

Important: DataDirect Bulk Load requires a licensed installation of the driver. If the driver is installed with an evaluation license, the bulk load feature is available for prototyping with your applications, but with limited scope. Contact your sales representative or DataDirect SupportLink for further information.

Important: Because a bulk load operation may bypass data integrity checks, your application must ensure that the data it is transferring does not violate integrity constraints in the database. For example, suppose you are bulk loading data into a database table and some of that data duplicates data stored as a primary key, which must be unique. The driver will not throw an exception to alert you to the error; your application must provide its own data integrity checks.

See also

Using a DDBulkLoad object on page 50
BulkLoadBatchSize on page 69

Using a DDBulkLoad object

To create a DDBulkLoad object, create an instance of the DDBulkLoadFactory class as shown in the following example.

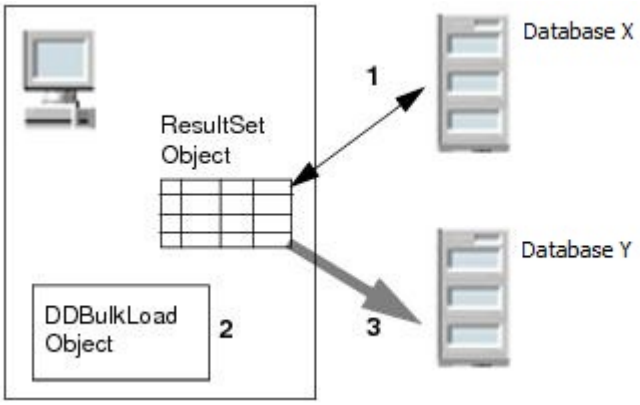
```
import com.ddtek.jdbc.extensions.*
// Get Database Connection
Connection con = DriverManager.getConnection(
   "jdbc:datadirect:redshift://server3:5439;
   DatabaseName=Test;User=admin;Password=secret");

// Get a DDBulkLoad object
DDBulkLoad bulkLoad = com.ddtek.jdbc.extensions.DDBulkLoadFactory.getInstance(con);
```

Once your application has created a DDBulkLoad object, you can use the DDBulkLoad methods to provide the driver with the following instructions:

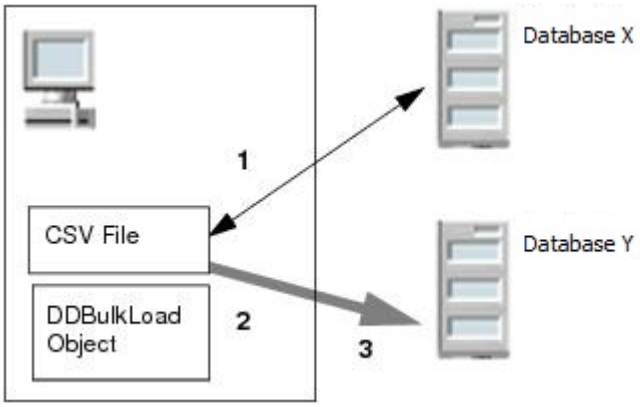
- The location where the driver obtains the data to load. The driver can obtain data from either of the following sources:
  - ResultSet object generated from a query.
  - Comma-separated value (CSV) file. The driver can export data from a table or ResultSet object into a CSV file, or the driver can use a CSV file created by another DataDirect Connect product.
- The location where the driver loads the data.

Suppose you need to migrate data from database system *X* to database *Y*. The following figure shows how your application would load the data from a ResultSet object created from a query. This scenario assumes that the driver is connected to both databases.



1. The application executes a query to the *X* database in the normal way and retrieves the results in a ResultSet object.
2. The application creates a DDBulkLoad object and instructs the driver to load data from the ResultSet object into the *Y* database.
3. The driver loads the data from the ResultSet object into the table.

The following figure shows how you would load the data using a CSV file instead of a ResultSet object. The CSV file is created by exporting data from a table on the *X* database.



1. The application creates a DDBulkLoad object.
2. The application specifies that the driver export the data from a table on database *X* into a CSV file.
3. The application instructs the driver to load data from the CSV file into the table on database *Y* table. The driver loads the data from the CSV file into the table on database *Y*.

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

See also

Exporting data to a CSV file on page 52
Loading Data from a ResultSet object on page 52
Loading data from a CSV file on page 52
Permissions for bulk load from a CSV file on page 54

Exporting data to a CSV file

Using the methods provided for bulk load, the driver can export data from either of the following sources into a CSV file:

- Database table. Use the setTableName() method, specifying the table name. For example, to export data from a table named GBMAXTABLE to a file named tmp.csv, you would specify:

  ```
  bulkLoad.setTableName("GBMAXTABLE");
  ```

  and

  ```
  bulkLoad.export("tmp.csv");
  ```

Note: Alternatively, you can create a file reference to the CSV file and use the export() method to specify the file reference:`File csvFile = new File ("tmp.csv");bulkLoad.export(csvFile);`

- ResultSet object. Create a file reference to a CSV file, and use the export() method, specifying the ResultSet object and the file reference. For example, to export data from a ResultSet Object named rs to a file named tmp.csv, you would specify:

  ```
  File csvFile = new File ("tmp.csv");
  bulkLoad.export(rs, csvFile);
  ```

If the CSV file does not already exist, the driver creates it when the export() method is executed. The driver also creates a bulk load configuration file, which describes the structure of the CSV file.

Loading Data from a ResultSet object

Use the setTableName() method to specify the table to load the data. Then, use the load() method, specifying the ResultSet object. For example, to load data from a ResultSet object named rs into a table named GBMAXTABLE, you would specify:

```
bulkLoad.setTableName("GBMAXTABLE");
```

and

```
bulkLoad.load(rs);
```

This example loads the first column in the result set to the ColName1 column, the second column in the result set to the ColName2 column, and so on.

Use the BulkLoadBatchSize property 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 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.

Loading data from a CSV file

Use the setTableName() method to specify the table to load the data. Then, use the load() method, specifying the CSV file. For example, to load data from a file named tmp.csv into a table named GBMAXTABLE, you would specify:

```
bulkLoad.setTableName("GBMAXTABLE");
```

and

```
bulkLoad.load("tmp.csv");
```

Alternatively, you can create a file reference to the CSV file, and use the load() method to specify the file reference:

```
File csvFile = new File("tmp.csv");
bulkLoad.load(csvFile);
```

Use the BulkLoadBatchSize property 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 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.

Specifying the bulk load operation

You can specify which type of bulk load operation will be performed when a load method is called by setting the operation property using the setProperties() method of the DDBulkLoad interface. The operation property accepts the following values: `insert`, `update`, `delete` and `upsert`. The default value is `insert`.

The following example changes the type of bulk load operation to update:

```
DDBulkLoad bulkLoad =
com.ddtek.jdbc.extensions.DDBulkLoadFactory.getInstance(connection);
Properties props = new Properties();
props.put("operation", "update");
bulkLoad.setProperties(props);
```

Logging

If logging is enabled for bulk load, a log file records information for each bulk load operation. Logging is enabled by specifying a file name and location for the log file using the setLogFile() method.

The log file records the following types of information about each bulk load operation:

- Total number of rows that were read
- Total number of rows that successfully loaded
- Total number of rows that failed to load

For example, the following log file shows that of 11 rows that were read, all successfully loaded:

```
/*----- Load Started: <Feb 25, 2009 11:20:09 AM EST>---------------------*/
Total number of rows read 11
Total number of rows successfully loaded 11
Total number of rows that failed to load 0
```

Example A: Enabling Logging on Windows

To enable logging using a log file named bulk_load.log located in the C:\temp directory, specify:

```
bulkLoad.setLogFile(C:\\temp\\bulk_load.log)
```

Note: 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: `C:\\temp\\bulk_load.log`.

Example B: Enabling Logging on UNIX/Linux

To enable logging using a log file named bulk_load.log located in the /tmp directory, specify:

```
bulkLoad.setLogFile(/tmp/bulk_load.log)
```

Permissions for bulk load from a CSV file

To bulk load data from a comma-separated value (CSV) file with the drivers, the application and driver code bases must be granted security permissions in the security policy file of the Java Platform as shown in the following examples.

```
grant codeBase "file:/install_dir/lib/60/-" {
   permission java.util.PropertyPermission "true", "read";
   permission java.util.PropertyPermission "file.encoding", "read";
   permission java.util.PropertyPermission "user.dir", "read";
   permission java.lang.RuntimePermission "readFileDescriptor";
};
```

Using CSV files

As described in "Exporting Data to a CSV File," the driver can create a CSV file by exporting data from a table or ResultSet object. For example, suppose you want to export data from a 4-column table named GBMAXTABLE into a CSV file. The contents of the CSV file, named GBMAXTABLE.csv, might look like the following example:

```
1,0x6263,"bc","bc"
2,0x636465,"cde","cde"
3,0x64656667,"defg","defg"
4,0x6566676869,"efghi","efghi"
5,0x666768696a6b,"fghijk","fghijk"
6,0x6768696a6b6c6d,"ghijklm","ghijklm"
7,0x68696a6b6c6d6e6f,"hijklmno","hijklmno"
8,0x696a6b6c6d6e6f7071,"ijklmnopq","ijklmnopq"
9,0x6a6b6c6d6e6f70717273,"jklmnopqrs","jklmnopqrs"
10,0x6b,"k","k"
```

Bulk load configuration file

Each time data is exported to a CSV file, a bulk load configuration file is created. This file has the same name as the CSV file, but with an .xml extension (for example, `GBMAXTABLE.xml`).

The bulk load configuration file defines in its metadata the names and data types of the columns in the CSV file based on those in the table or ResultSet object. It also defines other data properties, such as the source code page for character data types, precision and scale for numeric data types, and nullability for all data types. The format of `GBMAXTABLE.xml` might look like the following example.

```
<?xml version="1.0" encoding="utf-8"?>
<table codepage="UTF-8" xsi:noNamespaceSchemaLocation=
"http://media.datadirect.com/download/docs/ns/bulk/BulkData.xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<row>
   <column datatype="DECIMAL" precision="38" scale="0" nullable="false">
      INTEGERCOL</column>
   <column datatype="VARBINARY" length="10"
nullable="true">VARBINCOL</column>
```

```
   <column datatype="VARCHAR" length="10" sourcecodepage="Windows-1252"
      externalfilecodepage="Windows-1252"
nullable="true">VCHARCOL</column>
   <column datatype="VARCHAR" length="10" sourcecodepage="Windows-1252"
      externalfilecodepage="Windows-1252"
nullable="true">UNIVCHARCOL</column>
</row>
</table>
```

If the driver cannot read a bulk load configuration file (for example, because it was inadvertently deleted), the driver reads all data as character data. The character set used by the driver is UTF-8.

Bulk load configuration file schema

The bulk load configuration file must conform to the bulk load configuration XML schema defined at the following Web site:

http://media.datadirect.com/download/docs/ns/bulk/BulkData.xsd

The driver throws an exception if either of the following circumstances occur:

- If the driver performs a data export and the CSV file cannot be created
- If the driver performs a bulk load operation and the driver detects that the CSV file does not comply with the XML Schema described in the bulk load configuration file

Verifying the bulk load configuration file for database connections

Before performing a bulk load operation, your application can verify the metadata in the bulk load configuration file against the structure of the target table using the validateTableFromFile() method. This ensures that the data in the CSV file is compatible with the target table structure.

Because the verification does not check the actual data in the CSV file, it is possible that the load can fail even if the verification succeeds. For example, suppose your bulk load configuration file has an Integer column that matches an Integer column in the target database. If you modified the data for the Integer column in the CSV file to contain non-digit characters, the bulk load operation would fail even though a verification using validateTableFromFile() would succeed.

Not all error messages or warnings generated by the verification process mean that the load will fail. Some messages notify you about possible incompatibilities between the source and target tables. For example, if the CSV file has a column defined as an integer and the column in the target table is defined as smallint, the load may still succeed if the values in the source column are small enough to fit in a smallint column.

Character set conversions

When you export data from a database to a CSV file, the CSV file uses the same code page as the table from which the data was exported. If the CSV file and the target table use different code pages, performance for bulk load operations can suffer because the driver must perform a character set conversion.

To avoid character set conversions, your application can specify which code page to use for the CSV file when exporting data. You can specify any of the following code pages:

US_ASCII	IBM273	IBM01140
ISO_8859_1	IBM277	IBM01141
ISO_8859_2	IBM278	IBM01142
ISO_8859_3	IBM280	IBM01143
ISO_8859_4	IBM284	IBM01144
ISO_8859_5	IBM285	IBM01145
ISO_8859_6	IBM290	IBM01146
ISO_8859_7	IBM297	IBM01147
ISO_8859_8	IBM420	IBM01148
ISO_8859_9	IBM424	IBM01149
JIS_Encoding	IBM500	WINDOWS-1250
Shift_JIS	IBM851	WINDOWS-1251
EUC_JP	IBM855	WINDOWS-1252
KS_C_5601	IBM857	WINDOWS-1253
ISO_2022_KR	IBM860	WINDOWS-1254
EUC_KR	IBM861	WINDOWS-1255
ISO_2022_JP	IBM863	WINDOWS-1256
GB2312	IBM864	WINDOWS-1257
ISO_8859_13	IBM865	WINDOWS-1258
ISO_8859_15	IBM869	WINDOWS-854
GBK	IBM870	IBM-939
IBM850	IBM871	IBM-943_P14A-2000
IBM852	IBM1026	IBM-4396
IBM437	KOI8_R	IBM-5026
IBM862	HZ_GB_2312	IBM-5035
Big5	IBM866	UTF-8
MACINTOSH	IBM775	UTF-16LE
IBM037	IBM00858	UTF-16BE

For example, if the source database table uses a SHIFT-JIS code page and the target table uses a EUC-JP code page, specify `setCodePage("EUC_JP")` to ensure that the CSV file will use the same code page as the target table. If the code page you need to use is not listed, contact Technical Support to request support for that code page.

External overflow files

When you export data into a CSV file, you can choose to create one large file or multiple smaller files. For example, if you are exporting BLOB data that is a total of several GB, you may want to break the data that into multiple smaller files of 100 MB each.

If the values set by the setCharacterThreshold() or setBinaryThreshold() methods are exceeded, separate files are generated to store character or binary data, respectively. Overflow files are located in the same directory as the CSV file.

The format for overflow file names is:

```
CSV_file_name.xxxxxx.lob
```

where:

CSV_file_name is the name of the CSV file.

xxxxxx is a 6-digit number that increments an overflow file.

For example, if multiple overflow files are created for a CSV file named CSV1, the file names would look like this:

```
CSV1.000001.lob
CSV1.000002.lob
CSV1.000003.lob
...
```

If the overflow file contains character data, the code page used by the file is the code page specified in the bulk load configuration file for the CSV file.

Discard file

If the driver was unable to load rows into the database for a bulk load operation from a CSV file, it can record all the rows that were unable to be loaded into the database in the discard file. The contents of the discard file is in the same format as that of the CSV file. After fixing reported issues in the discard file, the bulk load can be reissued, using the discard file as the CSV file.

A discard file is created by specifying a file name and location for the discard file using the setDiscardFile() method.

Example A: Creating a Discard File on Windows

To create a discard file named discard.csv located in the `C:\temp` directory, specify:

```
bulkLoad.setDiscardFile(C:\\temp\\discard.csv)
```

Note: 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: `C:\\temp\\discard.csv`.

Example B: Creating a Discard File on UNIX/Linux

To create a discard file named discard.csv located in the /tmp directory, specify:

```
bulkLoad.setDiscardFile(/tmp/discard.csv)
```

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.

- Statement pooling properties
- Failover properties
- Bulk load properties
- Client information properties
- Additional properties

Required properties

The following table summarizes properties required for connecting.

Property	Data Source Method	Default
DatabaseName on page 76	`getDatabaseName()` `setDatabaseName(String)`	No default value
Password on page 89	`getPassword()` `setPassword(String)`	No default value
PortNumber on page 89	`getPortNumber()` `setPortNumber(int)`	5439
ServerName on page 95	`getServerName()` `setServerName(String)`	No default value
User on page 102	`getUser()` `setUser(String)`	No default value

Data encryption properties

The following table summarizes properties used in the implementation of SSL data encryption, including server and client authentication.

Property	Data Source Method	Default
CryptoProtocolVersion on page 74	`getCryptoProtocolVersion()` `setCryptoProtocolVersion(String)`	No default value
EncryptionMethod on page 77	`getEncryptionMethod()` `setEncryptionMethod(String)`	`noEncryption`
HostNameInCertificate on page 79	`getHostNameInCertificate()` `setHostNameInCertificate(String)`	No default value
KeyPassword on page 83	`getKeyPassword()` `setKeyPassword(String)`	No default value

Property	Data Source Method	Default
KeyStore on page 83	`getKeyStore()` `setKeyStore(String)`	No default value
KeyStorePassword on page 84	`getKeyStorePassword()` `setKeyStorePassword(String)`	No default value
TrustStore on page 100	`getTrustStore()` `setTrustStore(String)`	No default value
TrustStorePassword on page 101	`getTrustStorePassword()` `setTrustStorePassword(String)`	No default value
ValidateServerCertificate on page 104	`getValidateServerCertificate()` `setValidateServerCertificate(boolean)`	`true`

Proxy server properties

The following table summarizes proxy server connection properties.

Property	Data Source Method	Default
ProxyHost on page 91	`getProxyHost()` `setProxyHost(String)`	No default value
ProxyPassword on page 93	`getProxyPassword()` `setProxyPassword(String)`	No default value
ProxyPort on page 91	`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 92	`getProxyUser()` `setProxyUser(String)`	No default value
UseSystemProxyOptions on page 102	`getUseSystemProxy()` `setUseSystemProxy(Boolean)`	`true`

Data type properties

The following table summarizes connection properties which can be used to handle data types.

Table 3: Data Type Properties

Property	Data Source Method	Default
ConvertNull on page 74	`getConvertNull()` `setConvertNull(Boolean)`	`true`
JavaDoubleToString on page 82	`getJavaDoubleToString()` `setJavaDoubleToString(Boolean)`	`false`
MaxNumericPrecision on page 86	`getMaxNumericPrecision()` `setMaxNumericPrecision(Integer)`	`1000`
MaxNumericScale on page 86	`setMaxNumericScale()` `setMaxNumericScale(Integer)`	`998`
MaxVarcharSize on page 88	`getMaxVarcharSize()` `setMaxVarcharSize(Integer)`	`10485760`
VarcharClobThreshold on page 103	`getVarcharClobThreshold()` `setVarcharClobThreshold(Varchar)`	`32768`

Timeout properties

The following table summarizes timeout connection properties.

Property	Data Source Method	Default
EnableCancelTimeout on page 76	`getEnableCancelTimeout()` `setEnableCancelTimeout(Boolean)`	`false`
LoginTimeout	`getLoginTimeout()` `setLoginTimeout(Integer)`	`0`
QueryTimeout	`getQueryTimeout()` `setQueryTimeout(Integer)`	`0`

Statement pooling properties

The following table summarizes statement pooling connection properties.

Table 4: Statement Pooling Properties

Property	Data Source Method	Default
ImportStatementPool on page 80	`getImportStatementPool()` `setImportStatementPool(String)`	No default value

Property	Data Source Method	Default
MaxPooledStatements on page 87	`getMaxPooledStatements()` `setMaxPooledStatements(Integer)`	`0`
RegisterStatementPoolMonitorMBean on page 94	`getRegisterStatementPoolMonitorMBean()` `setRegisterStatementPoolMonitorMBean(Boolean)`	`0`

Bulk load properties

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

Property	Data Source Method	Default
BulkLoadBatchSize on page 69	`getBulkLoadBatchSize()` `setBulkLoadBatchSize(Long)`	`1000`

Failover properties

The following table summarizes connection properties which can be used to implement failover.

Property	Data Source Method	Default
ConnectionRetryCount on page 72	`getConnectionRetryCount()` `setConnectionRetryCount(Integer)`	`5`
ConnectionRetryDelay on page 73	`getConnectionRetryDelay()` `setConnectionRetryDelay(Integer)`	`1`

Client information properties

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

Property	Data Source Method	Default
AccountingInfo on page 66	`getAccountingInfo()` `setAccountingInfo(String)`	No default value
ApplicationName on page 67	`getApplicationName()` `setApplicationName(String)`	No default value
ClientHostName on page 71	`getClientHostName()` `setClientHostName(String)`	No default value

Property	Data Source Method	Default
ClientUser on page 71	`getClientUser()` `setClientUser(String)`	No default value
ProgramID on page 90	`getProgramID()` `setProgramID(String)`	No default value

Additional properties

The following table summarizes additional connection properties.

Property	Data Source Method	Default
BatchMechanism on page 68	`getBatchMechanism()` `setBatchMechanism(String)`	`MultiRowInsert`
CallEscapeBehavior on page 69	`getCallEscapeBehavior()` `setCallEscapeBehavior(String)`	`callIfNoReturn`
CatalogOptions on page 70	`getCatalogOptions()` `setCatalogOptions(Integer)`	`2`
ExtendedColumnMetadata on page 78	`getExtendedColumnMetadata()` `setExtendedColumnMetadata(Boolean)`	`false`
InitializationString on page 80	`getInitializationString()` `setInitializationString(String)`	No default value
InsensitiveResultSetBufferSize on page 81	`getInsensitiveResultsetBufferSize()` `setInsensitiveResultsetBufferSize(Integer)`	`2048`
ResultSetMetaDataOptions on page 95	`getResultSetMetaDataOptions()` `setResultSetMetaDataOptions(Integer)`	`0`
SpyAttributes on page 96	`getSpyAttributes()` `setSpyAttributes(String)`	No default value

Property	Data Source Method	Default
SupportsCatalogs on page 99	`getSupportsCatalogs()` `setSupportsCatalogs(Boolean)`	`true`
TransactionErrorBehavior on page 100	`getTransactionErrorBehavior()` `setTransactionErrorBehavior(String)`	`RollbackTransaction`

For details, see the following topics:

- AccountingInfo
- ApplicationName
- BatchMechanism
- BulkLoadBatchSize
- CallEscapeBehavior
- CatalogOptions
- ClientHostName
- ClientUser
- ConnectionRetryCount
- ConnectionRetryDelay
- ConvertNull
- CryptoProtocolVersion
- DatabaseName
- EnableCancelTimeout
- EncryptionMethod
- ExtendedColumnMetadata
- HostNameInCertificate
- ImportStatementPool
- InitializationString
- InsensitiveResultSetBufferSize
- JavaDoubleToString
- KeyPassword
- KeyStore
- KeyStorePassword
- LoginTimeout

- MaxNumericPrecision
- MaxNumericScale
- MaxPooledStatements
- MaxVarcharSize
- Password
- PortNumber
- ProgramID
- ProxyHost
- ProxyPort
- ProxyUser
- ProxyPassword
- QueryTimeout
- RegisterStatementPoolMonitorMBean
- ResultSetMetaDataOptions
- ServerName
- SpyAttributes
- SupportsCatalogs
- TransactionErrorBehavior
- TrustStore
- TrustStorePassword
- User
- UseSystemProxyOptions
- VarcharClobThreshold
- ValidateServerCertificate

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 Method

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

Default

No default value

Data Type

String

See also

Client information on page 48

ApplicationName

Purpose

Specifies the name of the application 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 the name of the application.

Data Source Method

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

Default

No default value

Data Type

String

See also

Client information on page 48

BatchMechanism

Purpose

Determines the mechanism that is used to execute batch operations.

Valid Values

`nativeBatch | multiRowInsert`

Behavior

If set to `nativeBatch`, the driver uses the data store's native batch mechanism to execute batch operations. NativeBatch returns individual update counts for each statement or parameter set in the batch.

If set to `multiRowInsert`, the driver uses a parameterized multi-row insert statement to execute batch inserts. MultiRowInsert returns only the total number of update counts in the batch, but provides substantial performance gains when performing batch inserts.

Notes

MultiRowInsert only applies to Insert statements for batch executes called with PreparedStatement objects. If a command other than the Insert statement is used, the driver uses the native batch mechanism to execute the command.

Data Source Method

```
public String getBatchMechanism()
```

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

Default

`multiRowInsert`

Data Type

String

See also

- Batch inserts and updates on page 49
- Performance considerations on page 41

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.

Data Source Method

```
public Long getBulkLoadBatchSize()
```

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

Default

```
1000
```

Data Type

Long

See also

- Batch inserts and updates on page 49
- DataDirect Bulk Load on page 50
- Performance considerations on page 41

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 Method

```
public Integer getCatalogOptions()
```

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

Default

2

Data Type

int

See also

- Performance considerations on page 41

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

ClientHostName

Purpose

Specifies the host name of the client machine 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 the host name of the client machine.

Data Source Method

```
public String getClientHostName()
```

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

Default

No default value

Data Type

String

ClientUser

Purpose

Specifies the user ID 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 valid user ID.

Data Source Method

```
public String getClientUser()
public void setClientUser(String)
```

Default

No default value

Data Type

String

ConnectionRetryCount

Purpose

The number of times the driver retries connection attempts to Redshift until a successful connection is established.

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 server to which it tried to connect.

Example

If this property is set to 2, the driver retries the server 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.

Data Source Methods

```
public Integer getConnectionRetryCount()
```

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

Default Value

5

Data Type

Integer

ConnectionRetryDelay

Purpose

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 and this property is set to 3, the driver retries the server 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 Value

1

Data Type

Integer

ConvertNull

Purpose

Controls how data conversions are handled for null values.

Valid Values

`true` | `false`

Behavior

If set to `true`, the driver checks the data type being 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`.

If set to `false`, 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.

Data Source Methods

```
public Boolean getConvertNull()
```

```
public void setConvertNull(Boolean)
```

Default Value

`true`

Data Type

Boolean

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 Method

```
public String getCryptoProtocolVersion()
```

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

Default

No default value

Data Type

String

See also

- Data Encryption on page 43
- EncryptionMethod on page 77

DatabaseName

Purpose

Specifies the name of the database to which you are connecting.

Valid Values

database_name

where:

database_name

is the name of a valid database.

Important: The value is case-insensistive if you have access privileges to query the list of databases on the server. If you do not have access, the value is case-sensitive.

Data Source Methods

```
public String getDatabaseName()
```

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

Default Value

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 Method

```
public Boolean getEnableCancelTimeout()
```

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

Default

`false`

Data Type

boolean

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 Method

```
public String getEncryptionMethod()
```

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

Default
noEncryption

Data Type
String

See also
- Data Encryption on page 43
- Performance considerations on page 41

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 Method

```
public String getHostNameInCertificate()
```

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

Default

No default value

Data Type

String

See also

Data Encryption on page 43

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.

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.

Notes

- If the driver cannot locate the specified file when establishing the connection, the connection fails and the driver throws an exception.
- For more information, refer to "Statement Pool Monitor" in the *Progress DataDirect for JDBC Drivers Reference*.

Data Source Methods

```
public String getImportStatementPool()
```

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

Default Value

No default value

Data Type

String

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:redshift://server1:5439;DatabaseName=test;
InitializationString=(command1;command2)
```

Data Source Method

```
public String getInitializationString()
```

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

Default

No default value

Data Type

String

InsensitiveResultSetBufferSize

Purpose

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

Valid Values

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

where:

x

is a positive integer that represents the amount of memory.

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 MB. If the size of the result set data exceeds available memory, then the driver pages the result set data to disk, which can have a negative performance effect. 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, then the driver pages the result set data to disk, which can have a negative performance effect. 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 Value

`2048`

Data Type

Integer

See also

- Performance considerations on page 41

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 Method

```
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 Method

```
public String getKeyPassword()
```

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

Default

No default vaue

Data Type

String

See also

Data Encryption on page 43

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.

Valid Values

string

where:

string

is a valid directory of a keystore file.

Notes

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

Data Source Method

```
public String getKeyStore()
```

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

Default

No default value

Data Type

String

See also

Data Encryption on page 43

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.

Valid Values

string

where:

string

is a valid password.

Notes

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

Data Source Method

```
public String getKeyStorePassword()
```

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

Default

No default value

Data Type

String

See also

Data Encryption on page 43

LoginTimeout

Purpose

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 specifies a number of seconds.

Behavior

If set to `0`, inactive connections are kept open.

If set to *x*, inactive connections are closed after the specified number of seconds passes.

Data Source Methods

```
public Integer getLoginTimeout()
```

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

Default Value

`0`

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 Method

```
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 Method

```
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.

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.

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.

Data Source Methods

```
public Integer getMaxPooledStatements()
```

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

Default Value

`0`

Data Type

Integer

See also

- Performance considerations on page 41

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

A password that is used to connect to the service.

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

Behavior

password

where:

password

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

Data Source Methods

`public String getPassword()`

`public void setPassword(String)`

Default Value

No default value

Data Type

String

See also

User ID/password authentication on page 13

PortNumber

Purpose

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

This property is supported only for data source connections.

Valid Values

port

where:

port

is the port number.

Data Source Method

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

Default

5439

Data Type

int

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 Method

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

Default

No default value

Data Type

String

See also

Client information on page 48

ProxyHost

Purpose

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 Value

No default value

Data Type

String

See also

Proxy server on page 14

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

See also

Proxy server on page 14

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

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 Value

No default value

Data Type

String

See also

Proxy server on page 14

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 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 that is set by this property, 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 Value

`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 87
- 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 Method

```
public Integer getResultSetMetaDataOptions()
```

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

Default

`0`

Data Type

int

See also

Performance Considerations

ServerName

Purpose

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 `MyRedshiftServer`

Note: When specifying IPv6 addresses in a connection URL or data source property, the address must be enclosed by brackets.

Data Source Method

```
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.

Behavior

Attribute	Description
`linelimit=`*`numberofchars`*	Sets the maximum number of characters that DataDirect Spy logs on a single line. The default is 0 (no maximum limit).
`log=(file)`*`filename`*	Directs logging to the file specified by *`filename`*. For Windows, if coding a path 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;logIS=yes;logTName=yes`

`log=(filePrefix)`*`file_prefix`*	Directs logging to a file prefixed by *`file_prefix`*. The log file is named *`file_prefixX`*`.log`
	where:
	X is an integer that increments by 1 for each connection on which the prefix is specified.
	For example, if the attribute `log=(filePrefix)C:\\temp\\spy_` is specified on multiple connections, the following logs are created:
	`C:\temp\spy_1.log`
	`C:\temp\spy_2.log`
	`C:\temp\spy_3.log`
	`...`
	If coding a path to the log file in a Java string, the backslash character (\) must be preceded by the Java escape character, a backslash.
	For example:
	`log=(filePrefix)C:\\temp\\spy_;logIS=yes;logTName=yes`

`log=System.out`	Directs logging to the Java output standard, `System.out`.

Attribute	Description
logIS={yes | no | nosingleread}	Specifies whether DataDirect Spy logs activity on InputStream and Reader objects. When logIS=nosingleread, logging on InputStream and Reader objects is active; however, logging of the single-byte read InputStream.read or single-character Reader.read is suppressed to prevent generating large log files that contain single-byte or single character read messages. The default is no.
logLobs={yes | no}	Specifies whether DataDirect Spy logs activity on BLOB and CLOB objects.
logTName={yes | no}	Specifies whether DataDirect Spy logs the name of the current thread. The default is no.
timestamp={yes | no}	Specifies whether a timestamp is included on each line of the DataDirect Spy log. The default is yes.

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.
- If a log file name does not include the .log extension, the driver automatically appends it. For example, a file named spy.jsp is renamed to spy.jsp.log by the driver.

Data Source Method

```
public String getSpyAttributes()
```

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

Default

No default value

Data Type

String

See also

Refer to "Tracking JDBC Calls with DataDirect Spy" in the *Progress DataDirect for JDBC Drivers Reference* for more information about using DataDirect Spy.

SupportsCatalogs

Purpose

Enables support for catalogs. While Amazon Redshift 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 Method

```
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 Amazon Redshift server does not allow any operations on the connection except for rolling back the transaction.

Valid Values

`none` | `RollbackTransaction`

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.

Data Source Method

```
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.

Valid Values

`string`

where:

`string`

is the directory of the truststore file.

Notes

- 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`.

Data Source Method

```
public String getTrustStore()
```

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

Default

No default value

Data Type

String

See also

Data Encryption on page 43

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.

Valid Values

`string`

where:

`string`

is a valid password for the truststore file.

Notes

- 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`.

Data Source Method

```
public String getTrustStorePassword()
```

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

Default

No default value

Data Type

String

See also

Data Encryption on page 43

User

Purpose

Specifies the user name that is used to connect to Redshift.

Valid Values

String

where:

String

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

Data Source Methods

```
public String getUser()
```

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

Default Value

No default value

Data Type

String

See also

User ID/password authentication on page 13

UseSystemProxyOptions

Purpose

Determines whether the driver attempts to use JVM system properties to configure proxy server settings by default.

Valid Values

`true` | `false`

Behavior

If set to `true`, the driver attempts to use the settings of the `http.proxyHost` and `http.proxyPort` JVM system properties. If no proxy server settings are configured on the JVM, the driver uses the properties specified.

If set to `false`, the driver uses only proxy server properties configured in the connection string or datasource. Specify this value when the driver is in a JVM environment with other Java applications, but you do not want to connect through the proxy server that is JVM system wide.

Data Source Methods

```
public Boolean getUseSystemProxy()
```

```
public void setUseSystemProxy(Boolean)
```

Default Value

`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 Method

```
public Varchar getVarcharClobThreshold()
```

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

Default

32768

Data Type

VARCHAR or LONGVARCHAR

See also

Performance considerations on page 41

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).

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.
- 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.

Data Source Method

```
public Boolean getValidateServerCertificate()
```

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

Default

`true`

Data Type

boolean

See also

Data Encryption on page 43

6

Supported SQL statements and extensions

The driver provides support for the SQL statements and the SQL extensions described in this section. SQL extensions are denoted by an (EXT) in the topic title.

For details, see the following topics:

Alter Session (EXT)

Purpose

Changes various attributes of a local or remote session. A local session maintains the state of the overall connection. A remote session maintains the state that pertains to a particular remote data source connection.

Syntax

```
ALTER SESSION SET attribute_name=value
```

where:

attribute_name

specifies the name of the attribute to be changed. Attributes apply to either local or remote sessions.

value

specifies the value for that attribute.

The following table lists the local and remote session attributes, and provides descriptions of each.

Table 5: Alter Session Attributes

Attribute Name	Session Type	Description
Current_Schema	Local	Sets the current schema for the local session. The current schema is the schema used when an identifier in a SQL statement is unqualified. The string value must be the name of a schema visible in the local session. For example: `ALTER SESSION SET CURRENT_SCHEMA=REDSHIFT`
Stmt_Call_Limit	Local	Sets the maximum number of Web service calls the driver can make in executing a statement. Setting the Stmt_Call_Limit attribute has the same effect as setting the Statement Call Limit connection option. It sets the default Web service call limit used by any statement on the connection. Executing this command on a statement overrides the previously set Statement Call Limit for the connection. The value specified must be a positive integer or `0`. The value `0` means that no call limit exists. For example: `ALTER SESSION SET STMT_CALL_LIMIT=150`

Ws_Call_Count	Remote	Resets the Web service call count of a remote session to the value specified. The value must be `0` or a positive integer. WS_Call_Count represents the total number of Web service calls made to the remote data source instance for the current session. For example:
		`ALTER SESSION SET redshift.WS_CALL_COUNT=0`
		The current value of WS_Call_Count can be obtained by referring to the System_Remote_Sessions system table (see SYSTEM_REMOTE_SESSIONS Catalog Table for details). For example:
		`SELECT * from information_schema.system_remote_sessions WHERE session_id = cursessionid()`

Delete

Purpose

The Delete statement is used to delete rows from a table.

Syntax

`DELETE FROM table_name [WHERE search_condition]`

where:

table_name

specifies the name of the table from which you want to delete rows.

search_condition

is an expression that identifies which rows to delete from the table.

Notes

- The Where clause determines which rows are to be deleted. Without a Where clause, all rows of the table are deleted, but the table is left intact. See "Where Clause" for information about the syntax of Where clauses. Where clauses can contain subqueries.

Example A

This example shows a Delete statement on the emp table.

```
DELETE FROM emp WHERE emp_id = 'E10001'
```

Each Delete statement removes every record that meets the conditions in the Where clause. In this case, every record having the employee ID `E10001` is deleted. Because employee IDs are unique in the employee table, at most, one record is deleted.

Example B

This example shows using a subquery in a Delete clause.

```
DELETE FROM emp WHERE dept_id = (SELECT dept_id FROM dept WHERE dept_name =
'Marketing')
```

The records of all employees who belong to the department named Marketing are deleted.

Notes

- To enable Insert, Update, and Delete, set the ReadOnly connection option to `false`.

Explain Plan

Purpose

Retrieves a detailed list of the elements in the execution plan. It generates a result set with a single column named `OPERATION`. The individual elements that comprise the plan are returned as rows in the result set.

Syntax

```
EXPLAIN PLAN FOR {SELECT ...}
```

The returned list of elements includes the indexes used for performing the query and can be used to optimize the query.

Insert

Purpose

The Insert statement is used to add new rows to a local table. You can specify either of the following options:

- List of values to be inserted as a new row
- Select statement that copies data from another table to be inserted as a set of new rows

Syntax

```
INSERT INTO table_name [(column_name[,column_name]...)] {VALUES (expression
[,expression]...) | select_statement}
```

table_name

is the name of the table in which you want to insert rows.

column_name

is optional and specifies an existing column. Multiple column names (a column list) must be separated by commas. A column list provides the name and order of the columns, the values of which are specified in the Values clause. If you omit a *column_name* or a column list, the value expressions must provide values for all columns defined in the table and must be in the same order that the columns are defined for the table. Table columns that do not appear in the column list are populated with the default value, or with NULL if no default value is specified.

expression

is the list of expressions that provides the values for the columns of the new record. Typically, the expressions are constant values for the columns. Character string values must be enclosed in single quotation marks (’). See "Literals" for more information.

select_statement

is a query that returns values for each *column_name* value specified in the column list. Using a Select statement instead of a list of value expressions lets you select a set of rows from one table and insert it into another table using a single Insert statement. The Select statement is evaluated

before any values are inserted. This query cannot be made on the table into which values are inserted. See "Select" for information about Select statements.

Notes

- To enable Insert, Update, and Delete, set the ReadOnly connection option to `false`.

Specifying an external ID column

Use the following syntax to specify an external ID column to look up the value of a foreign key column.

Syntax

column_name EXT_ID [*schema_name*.[*table_name*.]]*ext_id_column*

where:

EXT_ID

is used to specify that the column specified by *ext_id_column* is used to look up the rowid to be inserted into the column specified by *column_name*.

schema_name

is the name of the schema of the table that contains the foreign key column being specified as the external ID column.

table_name

is the name of the table that contains the foreign key column being specified as the external ID column.

ext_id_column

is the external ID column.

Example A

This example uses a list of expressions to insert records. Each Insert statement adds one record to the table. In this case, one record is added to the table emp. Values are specified for five columns. The remaining columns in the table are assigned the default value or NULL if no default value is specified.

```
INSERT INTO emp (last_name,
                 first_name,
                 emp_id,
                 salary,
                 hire_date)
VALUES ('Smith', 'John', 'E22345', 27500, {1999-04-06})
```

Example B

This example uses a Select statement to insert records. The number of columns in the result of the Select statement must match exactly the number of columns in the table if no column list is specified, or it must match the number of column names specified in the column list. A new entry is created in the table for every row of the Select result.

```
INSERT INTO emp1 (first_name,
                  last_name,
```

```
                  emp_id,
                  dept,
                  salary)
SELECT first_name, last_name, emp_id, dept, salary FROM emp
WHERE dept = 'D050'
```

Example C

This example uses a list of expressions to insert records and specifies an external ID column (a foreign key column) named `accountId` that references a table that has an external ID column named `AccountNum`.

```
INSERT INTO emp (last_name,
                    first_name,
                    emp_id,
                    salary,
                    hire_date,
                    accountId EXT_ID AccountNum)
VALUES ('Smith', 'John', 'E22345', 27500, {1999-04-06}, 0001)
```

Select

Purpose

Use the Select statement to fetch results from one or more tables.

Syntax

```
SELECT select_clause from_clause
[where_clause]
[groupby_clause]
[having_clause]
[{UNION [ALL | DISTINCT] |
 {MINUS [DISTINCT] | EXCEPT [DISTINCT]} |
 INTERSECT [DISTINCT]} select_statement]
[limit_clause]
```

where:

select_clause

specifies the columns from which results are to be returned by the query. See "Select clause" for a complete explanation.

from_clause

specifies one or more tables on which the other clauses in the query operate. See "From clause" for a complete explanation.

where_clause

is optional and restricts the results that are returned by the query. See "Where clause" for a complete explanation.

groupby_clause

is optional and allows query results to be aggregated in terms of groups. See "Group By clause" for a complete explanation.

having_clause

is optional and specifies conditions for groups of rows (for example, display only the departments that have salaries totaling more than $200,000). See "Having clause" for a complete explanation.

UNION

is an optional operator that combines the results of the left and right Select statements into a single result. See "Union operator" for a complete explanation.

INTERSECT

is an optional operator that returns a single result by keeping any distinct values from the results of the left and right Select statements. See "Intersect operator" for a complete explanation.

EXCEPT | MINUS

are synonymous optional operators that returns a single result by taking the results of the left Select statement and removing the results of the right Select statement. See "Except and Minus operators" for a complete explanation.

orderby_clause

is optional and sorts the results that are returned by the query. See "Order By clause" for a complete explanation.

limit_clause

is optional and places an upper bound on the number of rows returned in the result. See "Limit clause" for a complete explanation.

Select clause

Purpose

Use the Select clause to specify with a list of column expressions that identify columns of values that you want to retrieve or an asterisk (*) to retrieve the value of all columns.

Syntax

```
SELECT [{LIMIT offsetnumber | TOP number}] [ALL | DISTINCT] {* | column_expression
[[AS] column_alias] [,column_expression [[AS] column_alias], ...]}
```

where:

LIMIT *offset number*

creates the result set for the Select statement first and then discards the first number of rows specified by *offset* and returns the number of remaining rows specified by *number*. To not discard any of the rows, specify 0 for *offset*, for example, LIMIT 0 *number*. To discard the first *offset* number of rows and return all the remaining rows, specify 0 for *number*, for example, LIMIT *offset*0.

`TOP number`

is equivalent to `LIMIT 0number`.

`column_expression`

can be simply a column name (for example, `last_name`). More complex expressions may include mathematical operations or string manipulation (for example, `salary * 1.05`). See "SQL expressions" for details. *`column_expression`* can also include aggregate functions. See "Aggregate functions" for details.

`column_alias`

can be used to give the column a descriptive name. For example, to assign the alias department to the column dep:

```
SELECT dep AS department FROM emp
```

`DISTINCT`

eliminates duplicate rows from the result of a query. This operator can precede the first column expression. For example:

```
SELECT DISTINCT dep FROM emp
```

Notes

- Separate multiple column expressions with commas (for example, `SELECT last_name, first_name, hire_date`).
- Column names can be prefixed with the table name or table alias. For example, `SELECT emp.last_name` or `e.last_name`, where `e` is the alias for the table `emp`.
- NULL values are not treated as distinct from each other. The default behavior is that all result rows be returned, which can be made explicit with the keyword `ALL`.

See also

SQL expressions on page 125

Aggregate functions

Aggregate functions can also be a part of a Select clause. Aggregate functions return a single value from a set of rows. An aggregate can be used with a column name (for example, `AVG(salary)`) or in combination with a more complex column expression (for example, `AVG(salary * 1.07)`).

The following table lists supported aggregate functions.

Note: Doubly nested aggregates, such as `SUM(COUNT(col1))`, are currently not permitted by the driver.

Table 6: Aggregate Functions

Aggregate	Returns
AVG	The average of the values in a numeric column expression. For example, `AVG(salary)` returns the average of all salary column values.

COUNT	The number of values in any field expression. For example, `COUNT(name)` returns the number of name values. When using `COUNT` with a field name, `COUNT` returns the number of non-NULL column values. A special example is `COUNT(*)`, which returns the number of rows in the set, including rows with NULL values. **Note:** The driver does not support `COUNT(DISTINCT *)`. For example, `SELECT COUNT(DISTINCT *) FROM mytable` results in a syntax error.
MAX	The maximum value in any column expression. For example, `MAX(salary)` returns the maximum salary column value.
MIN	The minimum value in any column expression. For example, `MIN(salary)` returns the minimum salary column value.
SUM	The total of the values in a numeric column expression. For example, `SUM(salary)` returns the sum of all salary column values.

Example

The following example uses the `COUNT`, `MAX`, and `AVG` aggregate functions:

```
SELECT
     COUNT(amount) AS numOpportunities,
     MAX(amount) AS maxAmount,
     AVG(amount) AS avgAmount
FROM opportunity o INNER JOIN user u
     ON o.ownerId = u.id
WHERE o.isClosed = 'false' AND
     u.name = 'MyName'
```

From clause

Purpose

The From clause indicates the tables to be used in the Select statement.

Syntax

`FROM` *`table_name`* `[`*`table_alias`*`] [,...]`

where:

`table_name`

is the name of a table or a subquery. Multiple tables define an implicit inner join among those tables. Multiple table names must be separated by a comma. For example:

```
SELECT * FROM emp, dep
```

Subqueries can be used instead of table names. Subqueries must be enclosed in parentheses. See "Subquery in a From clause" for an example.

table_alias

is a name used to refer to a table in the rest of the Select statement. When you specify an alias for a table, you can prefix all column names of that table with the table alias.

Example

The following example specifies two table aliases, e for emp and d for dep:

```
SELECT e.name, d.deptName
FROM emp e, dep d
WHERE e.deptId = d.id
```

table_alias is a name used to refer to a table in the rest of the Select statement. When you specify an alias for a table, you can prefix all column names of that table with the table alias. For example, given the table specification:

```
FROM emp E
```

you may refer to the last_name field as E.last_name. Table aliases must be used if the Select statement joins a table to itself. For example:

```
SELECT * FROM emp E, emp F WHERE E.mgr_id = F.emp_id
```

The equal sign (=) includes only matching rows in the results.

Join in a From clause

Purpose

You can use a Join as a way to associate multiple tables within a Select statement. Joins may be either explicit or implicit. For example, the following is the example from the previous section restated as an explicit inner join:

```
SELECT * FROM emp INNER JOIN dep ON id=empId
SELECT e.name, d.deptName
FROM emp e INNER JOIN dep d ON e.deptId = d.id;
```

whereas the following is the same statement as an implicit inner join:

```
SELECT * FROM emp, dep WHERE emp.deptID=dep.id
```

Note: The ON clause in a join expression must evaluate to a true or false value.

Syntax

```
FROM table_name {RIGHT OUTER | INNER | LEFT OUTER | CROSS | FULL OUTER} JOIN table.key
 ON search-condition
```

Example

In this example, two tables are joined using LEFT OUTER JOIN. T1, the first table named includes nonmatching rows.

```
SELECT * FROM T1 LEFT OUTER JOIN T2 ON T1.key = T2.key
```

If you use a CROSS JOIN, no ON expression is allowed for the join.

Subquery in a From clause

Subqueries can be used in the From clause in place of table references (*table_name*).

Example

```
SELECT * FROM (SELECT * FROM emp WHERE sal > 10000) new_emp, dept WHERE
new_emp.deptno = dept.deptno
```

See also

Subqueries on page 123

Where clause

Purpose

Specifies the conditions that rows must meet to be retrieved.

Syntax

```
WHERE expr1 rel_operator expr2
```

where:

expr1

is either a column name, literal, or expression.

expr2

is either a column name, literal, expression, or subquery. Subqueries must be enclosed in parentheses.

rel_operator

is the relational operator that links the two expressions.

Example

The following Select statement retrieves the first and last names of employees that make at least $20,000.

```
SELECT last_name, first_name FROM emp WHERE salary >= 20000
```

See also

SQL expressions on page 125
Subqueries on page 123

Group By clause

Purpose

Specifies the names of one or more columns by which the returned values are grouped. This clause is used to return a set of aggregate values.

Syntax

```
GROUP BY column_expression [,...]
```

where:

column_expression

is either a column name or a SQL expression. Multiple values must be separated by a comma. If *column_expression* is a column name, it must match one of the column names specified in the Select clause. Also, the Group By clause must include all non-aggregate columns specified in the Select list.

Example

The following example totals the salaries in each department:

```
SELECT dept_id, sum(salary) FROM emp GROUP BY dept_id
```

This statement returns one row for each distinct department ID. Each row contains the department ID and the sum of the salaries of the employees in the department.

See also

SQL expressions on page 125
Subqueries on page 123

Having clause

Purpose

Specifies conditions for groups of rows (for example, display only the departments that have salaries totaling more than $200,000). This clause is valid only if you have already defined a Group By clause.

Syntax

```
HAVING expr1 rel_operator expr2
```

where:

expr1 | expr2

can be column names, constant values, or expressions. These expressions do not have to match a column expression in the Select clause. See "SQL expressions" for details regarding SQL expressions.

rel_operator

is the relational operator that links the two expressions.

Example

The following example returns only the departments that have salaries totaling more than $200,000:

```
SELECT dept_id, sum(salary) FROM emp GROUP BY dept_id HAVING sum(salary) > 200000
```

See also

SQL expressions on page 125
Subqueries on page 123

Union operator

Purpose

Combines the results of two Select statements into a single result. The single result is all the returned rows from both Select statements. By default, duplicate rows are not returned. To return duplicate rows, use the All keyword (`UNION ALL`).

Syntax

```
select_statement
UNION [ALL | DISTINCT] | {MINUS [DISTINCT] | EXCEPT [DISTINCT]} | INTERSECT
[DISTINCT]select_statement
```

Notes

- When using the Union operator, the Select lists for each Select statement must have the same number of column expressions with the same data types and must be specified in the same order.

Example A

The following example has the same number of column expressions, and each column expression, in order, has the same data type.

```
SELECT last_name, salary, hire_date FROM emp
UNION
SELECT name, pay, birth_date FROM person
```

Example B

The following example is *not* valid because the data types of the column expressions are different (`salary FROM emp` has a different data type than `last_name FROM raises`). This example does have the same number of column expressions in each Select statement but the expressions are not in the same order by data type.

```
SELECT last_name, salary FROM emp
UNION
SELECT salary, last_name FROM raises
```

Intersect operator

Purpose

Intersect operator returns a single result set. The result set contains rows that are returned by both Select statements. Duplicates are returned unless the Distinct operator is added.

Syntax

```
select_statement
INTERSECT [DISTINCT]
select_statement
```

where:

`DISTINCT`

eliminates duplicate rows from the results.

Notes

- When using the Intersect operator, the Select lists for each Select statement must have the same number of column expressions with the same data types and must be specified in the same order.

Example A

The following example has the same number of column expressions, and each column expression, in order, has the same data type.

```
SELECT last_name, salary, hire_date FROM emp
INTERSECT [DISTINCT]
SELECT name, pay, birth_date FROM person
```

Example B

The following example is *not* valid because the data types of the column expressions are different (`salary FROM emp` has a different data type than `last_name FROM raises`). This example does have the same number of column expressions in each Select statement but the expressions are not in the same order by data type.

```
SELECT last_name, salary FROM emp
INTERSECT
SELECT salary, last_name FROM raises
```

Except and Minus operators

Purpose

Return the rows from the left Select statement that are not included in the result of the right Select statement.

Syntax

```
select_statement
{EXCEPT [DISTINCT] | MINUS [DISTINCT]}
select_statement
```

where:

`DISTINCT`

eliminates duplicate rows from the results.

Notes

- When using one of these operators, the Select lists for each Select statement must have the same number of column expressions with the same data types and must be specified in the same order.

Example A

The following example has the same number of column expressions, and each column expression, in order, has the same data type.

```
SELECT last_name, salary, hire_date FROM emp
EXCEPT
SELECT name, pay, birth_date FROM person
```

Example B

The following example is *not* valid because the data types of the column expressions are different (`salary FROM emp` has a different data type than `last_name FROM raises`). This example does have the same number of column expressions in each Select statement but the expressions are not in the same order by data type.

```
SELECT last_name, salary FROM emp
EXCEPT
SELECT salary, last_name FROM raises
```

Order By clause

Purpose

The Order By clause specifies how the rows are to be sorted.

Syntax

`ORDER BY` *`sort_expression`* `[DESC | ASC] [,...]`

where:

`sort_expression`

is either the name of a column, a column alias, a SQL expression, or the positioned number of the column or expression in the select list to use.

The default is to perform an ascending (ASC) sort.

Example

To sort by `last_name` and then by `first_name`, you could use either of the following Select statements:

```
SELECT emp_id, last_name, first_name FROM emp
ORDER BY last_name, first_name
```

or

```
SELECT emp_id, last_name, first_name FROM emp
ORDER BY 2,3
```

In the second example, `last_name` is the second item in the Select list, so `ORDER BY 2,3` sorts by `last_name` and then by `first_name`.

See also

SQL expressions on page 125

Limit clause

Purpose

Places an upper bound on the number of rows returned in the result.

Syntax

`LIMIT` *`number_of_rows`* `[OFFSET` *`offset_number`*`]`

where:

`number_of_rows`

specifies a maximum number of rows in the result. A negative number indicates no upper bound.

`OFFSET`

specifies how many rows to skip at the beginning of the result set. *`offset_number`* is the number of rows to skip.

Notes

- In a compound query, the Limit clause can appear only on the final Select statement. The limit is applied to the entire query, not to the individual Select statement to which it is attached.

Example

The following example returns a maximum of 20 rows.

```
SELECT last_name, first_name FROM emp WHERE salary > 20000 ORDER BY dept_idc LIMIT
20
```

Update

Purpose

An Update statement changes the value of columns in the selected rows of a table.

Syntax

`UPDATE` *`table_name`* `SET` *`column_name = expression`*

`[,` *`column_name = expression`*`] [WHERE` *`conditions`*`]`

`table_name`

is the name of the table for which you want to update values.

`column_name`

is the name of a column, the value of which is to be changed. Multiple column values can be changed in a single statement.

expression

is the new value for the column. The expression can be a constant value or a subquery that returns a single value. Subqueries must be enclosed in parentheses.

Example A

The following example changes every record that meets the conditions in the Where clause. In this case, the salary and exempt status are changed for all employees having the employee ID `E10001`. Because employee IDs are unique in the emp table, only one record is updated.

```
UPDATE emp SET salary=32000, exempt=1
WHERE emp_id = 'E10001'
```

Example B

The following example uses a subquery. In this example, the salary is changed to the average salary in the company for the employee having employee ID `E10001`.

```
UPDATE emp SET salary = (SELECT avg(salary) FROM emp)
WHERE emp_id = 'E10001'
```

Notes

- To enable Insert, Update, and Delete, set the ReadOnly connection option to `false`.
- A Where clause can be used to restrict which rows are updated.

Subqueries

A query is an operation that retrieves data from one or more tables or views. In this reference, a top-level query is called a Select statement, and a query nested within a Select statement is called a subquery.

A subquery is a query expression that appears in the body of another expression such as a Select, an Update, or a Delete statement. In the following example, the second Select statement is a subquery:

```
SELECT * FROM emp WHERE deptno IN (SELECT deptno FROM dept)
```

IN predicate

Purpose

The In predicate specifies a set of values against which to compare a result set. If the values are being compared against a subquery, only a single column result set is returned.

Syntax

value [NOT] IN (*value1, value2,...*)

OR

value [NOT] IN (*subquery*)

Example

```
SELECT * FROM emp WHERE deptno IN
```

```
(SELECT deptno FROM dept WHERE dname <> 'Sales')
```

EXISTS predicate

Purpose

The Exists predicate is true only if the cardinality of the subquery is greater than 0; otherwise, it is false.

Syntax

```
EXISTS (subquery)
```

Example

```
SELECT empno, ename, deptno FROM emp e WHERE EXISTS
(SELECT deptno FROM dept WHERE e.deptno = dept.deptno)
```

UNIQUE predicate

Purpose

The Unique predicate is used to determine whether duplicate rows exist in a virtual table (one returned from a subquery).

Syntax

```
UNIQUE (subquery)
```

Example

```
SELECT * FROM dept d WHERE UNIQUE
(SELECT deptno FROM emp e WHERE e.deptno = d.deptno)
```

Correlated subqueries

Purpose

A correlated subquery is a subquery that references a column from a table referred to in the parent statement. A correlated subquery is evaluated once for each row processed by the parent statement. The parent statement can be a Select, Update, or Delete statement.

A correlated subquery answers a multiple-part question in which the answer depends on the value in each row processed by the parent statement. For example, you can use a correlated subquery to determine which employees earn more than the average salaries for their departments. In this case, the correlated subquery specifically computes the average salary for each department.

Syntax

```
SELECT select_list
   FROM table1 t_alias1
   WHERE expr rel_operator
```

```
   (SELECT column_list
       FROM table2 t_alias2
       WHERE t_alias1.columnrel_operatort_alias2.column)
UPDATE table1 t_alias1
   SET column =
   (SELECT expr
       FROM table2 t_alias2
        WHERE t_alias1.column = t_alias2.column)
DELETE FROM table1 t_alias1
   WHERE column rel_operator
   (SELECT expr
       FROM table2 t_alias2
       WHERE t_alias1.column = t_alias2.column)
```

Notes

- Correlated column names in correlated subqueries must be explicitly qualified with the table name of the parent.

Example A

The following statement returns data about employees whose salaries exceed their department average. This statement assigns an alias to emp, the table containing the salary information, and then uses the alias in a correlated subquery:

```
SELECT deptno, ename, sal FROM emp x WHERE sal >
  (SELECT AVG(sal) FROM emp WHERE x.deptno = deptno)
  ORDER BY deptno
```

Example B

This is an example of a correlated subquery that returns row values:

```
SELECT * FROM dept "outer" WHERE 'manager' IN
  (SELECT managername FROM emp
  WHERE "outer".deptno = emp.deptno)
```

Example C

This is an example of finding the department number (deptno) with multiple employees:

```
SELECT * FROM dept main WHERE 1 <
  (SELECT COUNT(*) FROM emp WHERE deptno = main.deptno)
```

Example D

This is an example of correlating a table with itself:

```
SELECT deptno, ename, sal FROM emp x WHERE sal >
  (SELECT AVG(sal) FROM emp WHERE x.deptno = deptno)
```

SQL expressions

An expression is a combination of one or more values, operators, and SQL functions that evaluate to a value. You can use expressions in the Where, and Having of Select statements; and in the Set clauses of Update statements.

Expressions enable you to use mathematical operations as well as character string manipulation operators to form complex queries.

The driver supports both unquoted and quoted identifiers. An unquoted identifier must start with an ASCII alpha character and can be followed by zero

Quoted identifiers must be enclosed in double quotation marks (""). A quoted identifier can contain any Unicode character including the space character. The driver recognizes the Unicode escape sequence \uxxxx as a Unicode character. You can specify a double quotation mark in a quoted identifier by escaping it with a double quotation mark.

The maximum length of both quoted and unquoted identifiers is 128 characters.

Valid expression elements are:

- Column names
- Literals
- Operators
- Functions

Column names

The most common expression is a simple column name. You can combine a column name with other expression elements.

Literals

Literals are fixed data values. For example, in the expression `PRICE * 1.05`, the value 1.05 is a constant. Literals are classified into types, including the following:

- Binary
- Character string
- Date
- Floating point
- Integer
- Numeric
- Time
- Timestamp

The following table describes the literal format for supported SQL data types.

Table 7: Literal Syntax Examples

SQL Type	Literal Syntax	Example
BIGINT	*n* where *n* is any valid integer value in the range of the INTEGER data type	`12 or -34 or 0`

SQL Type	Literal Syntax	Example
BOOLEAN	Min Value: 0 Max Value: 1	`0` `1`
DATE	DATE'*date*'	`'2010-05-21'`
DATETIME	TIMESTAMP'*ts*'	`'2010-05-21 18:33:05.025'`

DECIMAL	*n.f*
	where:
	n
	is the integral part
	f
	is the fractional part
		`0.25`
		`3.1415`
		`-7.48`

DOUBLE	*n.f*E*x* where: *n* is the integral part *f* is the fractional part *x* is the exponent	`1.2E0` or `2.5E40` or `-3.45E2` or `5.67E-4`
INTEGER	*n* where *n* is a valid integer value in the range of the INTEGER data type	`12` or `-34` or `0`
LONGVARBINARY	'*hex_value*'	`'000482ff'`
LONGVARCHAR	'*value*'	`'This is a string literal'`
TIME	TIME'*time*'	`'2010-05-21 18:33:05.025'`
VARCHAR	'*value*'	`'This is a string literal'`

Character string literals

Text specifies a character string literal. A character string literal must be enclosed in single quotation marks. To represent one single quotation mark within a literal, you must enter two single quotation marks. When the data in the fields is returned to the client, trailing blanks are stripped.

A character string literal can have a maximum length of 32 KB, that is, (32*1024) bytes.

Example

```
'Hello'
'Jim''s friend is Joe'
```

Numeric literals

Unquoted numeric values are treated as numeric literals. If the unquoted numeric value contains a decimal point or exponent, it is treated as a real literal; otherwise, it is treated as an integer literal.

Example

```
+1894.1204
```

Binary literals

Binary literals are represented with single quotation marks. The valid characters in a binary literal are 0-9, a-f, and A-F.

Example

```
'00af123d'
```

Date/Time literals

Date and time literal values are enclosed in single quotion marks ('*value*').

- The format for a Date literal is DATE'*date*'.
- The format for a Time literal is TIME'*time*'.
- The format for a Timestamp literal is TIMESTAMP'*ts*'.

Integer literals

Integer literals are represented by a string of numbers that are not enclosed in quotation marks and do not contain decimal points.

Notes

- Integer constants must be whole numbers; they cannot contain decimals.
- Integer literals can start with sign characters (+/-).

Example

```
1994 or -2
```

Operators

This section describes the operators that can be used in SQL expressions.

Note: Numeric operators are restricted to numeric types. Numeric operators do not support non-numeric types.

Unary operator

A unary operator operates on only one operand.

operator operand

Binary operator

A binary operator operates on two operands.

operand1 operator operand2

If an operator is given a null operand, the result is always null. The only operator that does not follow this rule is concatenation (||).

Arithmetic operators

You can use an arithmetic operator in an expression to negate, add, subtract, multiply, and divide numeric values. The result of this operation is also a numeric value. The + and - operators are also supported in date/time fields to allow date arithmetic. The following table lists the supported arithmetic operators.

Table 8: Arithmetic Operators

Operator	Purpose	Example
+ -	Denotes a positive or negative expression. These are unary operators.	`SELECT * FROM emp WHERE comm = -1`
* /	Multiplies, divides. These are binary operators.	`UPDATE emp SET sal = sal + sal * 0.10`
+ -	Adds, subtracts. These are binary operators.	`SELECT sal + comm FROM emp WHERE empno > 100`

Concatenation operator

The concatenation operator manipulates character strings. The following table lists the only supported concatenation operator.

Table 9: Concatenation Operator

Operator	Purpose	Example
||	Concatenates character strings.	`SELECT 'Name is' || ename FROM emp`

The result of concatenating two character strings is the data type VARCHAR.

Comparison operators

Comparison operators compare one expression to another. The result of such a comparison can be TRUE, FALSE, or UNKNOWN (if one of the operands is NULL). The driver considers the UNKNOWN result as FALSE.

The following table lists the supported comparison operators.

Table 10: Comparison Operators

Operator	Purpose	Example
=	Equality test.	`SELECT * FROM emp WHERE sal = 1500`
!=<>	Inequality test.	`SELECT * FROM emp WHERE sal != 1500`
><	"Greater than" and "less than" tests.	`SELECT * FROM emp WHERE sal > 1500 SELECT * FROM emp WHERE sal < 1500`
>=<=	"Greater than or equal to" and "less than or equal to" tests.	`SELECT * FROM emp WHERE sal >= 1500 SELECT * FROM emp WHERE sal <= 1500`
LIKE	% and _ wildcards can be used to search for a pattern in a column. The percent sign denotes zero, one, or multiple characters, while the underscore denotes a single character. The right-hand side of a LIKE expression must evaluate to a string or binary.	`SELECT * FROM emp WHERE ENAME LIKE 'J%'`

ESCAPE clause in LIKE operator
LIKE 'pattern string' ESCAPE 'c'
	The Escape clause is supported in the LIKE predicate to indicate the escape character. Escape characters are used in the pattern string to indicate that any wildcard character that is after the escape character in the pattern string should be treated as a regular character.
	The default escape character is backslash (\).
		`SELECT * FROM emp WHERE ENAME LIKE 'J%\_%' ESCAPE '\'`
		This matches all records with names that start with letter 'J' and have the '_' character in them.
		`SELECT * FROM emp WHERE ENAME LIKE 'JOE\_JOHN' ESCAPE '\'`
		This matches only records with name 'JOE_JOHN'.

[NOT] IN	"Equal to any member of" test.	`SELECT * FROM emp WHERE job IN ('CLERK','ANALYST') SELECT * FROM emp WHERE sal IN (SELECT sal FROM emp WHERE deptno = 30)`
[NOT] BETWEEN x AND y	"Greater than or equal to x" and "less than or equal to y."	`SELECT * FROM emp WHERE sal BETWEEN 2000 AND 3000`

Operator	Purpose	Example
EXISTS	Tests for existence of rows in a subquery.	`SELECT empno, ename, deptno FROM emp e WHERE EXISTS (SELECT deptno FROM dept WHERE e.deptno = dept.deptno)`
IS [NOT] NULL	Tests whether the value of the column or expression is NULL.	`SELECT * FROM emp WHERE ename IS NOT NULL SELECT * FROM emp WHERE ename IS NULL`

Logical operators

A logical operator combines the results of two component conditions to produce a single result or to invert the result of a single condition. The following table lists the supported logical operators.

Table 11: Logical Operators

Operator	Purpose	Example
NOT	Returns TRUE if the following condition is FALSE. Returns FALSE if it is TRUE. If it is UNKNOWN, it remains UNKNOWN.	`SELECT * FROM emp WHERE NOT (job IS NULL)` `SELECT * FROM emp WHERE NOT (sal BETWEEN 1000 AND 2000)`
AND	Returns TRUE if both component conditions are TRUE. Returns FALSE if either is FALSE; otherwise, returns UNKNOWN.	`SELECT * FROM emp WHERE job = 'CLERK' AND deptno = 10`
OR	Returns TRUE if either component condition is TRUE. Returns FALSE if both are FALSE; otherwise, returns UNKNOWN.	`SELECT * FROM emp WHERE job = 'CLERK' OR deptno = 10`

Example

In the Where clause of the following Select statement, the AND logical operator is used to ensure that managers earning more than $1000 a month are returned in the result:

```
SELECT * FROM emp WHERE jobtitle = manager AND sal > 1000
```

Operator precedence

As expressions become more complex, the order in which the expressions are evaluated becomes important. The following table shows the order in which the operators are evaluated. The operators in the first line are evaluated first, then those in the second line, and so on. Operators in the same line are evaluated left to right in the expression.You can change the order of precedence by using parentheses. Enclosing expressions in parentheses forces them to be evaluated together.

Table 12: Operator Precedence

Precedence	Operator
1	+ (Positive), - (Negative)
2	*(Multiply), / (Division)
3	+ (Add), - (Subtract)
4	|| (Concatenate)
5	=, >, <, >=, <=, <>, != (Comparison operators)
6	NOT, IN, LIKE
7	AND
8	OR

Example A

The query in the following example returns employee records for which the department number is 1 or 2 and the salary is greater than $1000:

```
SELECT * FROM emp WHERE (deptno = 1 OR deptno = 2) AND sal > 1000
```

Because parenthetical expressions are forced to be evaluated first, the OR operation takes precedence over AND.

Example B

In the following example, the query returns records for all the employees in department 1, but only employees whose salary is greater than $1000 in department 2.

```
SELECT * FROM emp WHERE deptno = 1 OR deptno = 2 AND sal > 1000
```

The AND operator takes precedence over OR, so that the search condition in the example is equivalent to the expression `deptno = 1 OR (deptno = 2 AND sal > 1000)`.

Functions

The driver supports a number of functions that you can use in expressions, including String, Numeric, Timedate, and System functions.

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

Conditions

A condition specifies a combination of one or more expressions and logical operators that evaluates to either TRUE, FALSE, or UNKNOWN. You can use a condition in the Where clause of the Delete, Select, and Update statements; and in the Having clauses of Select statements. The following describes supported conditions.

Table 13: Conditions

Condition	Description
Simple comparison	Specifies a comparison with expressions or subquery results. `= , != , <>, < , >, <=, <=`
Group comparison	Specifies a comparison with any or all members in a list or subquery. `[= , != , <>, < , >, <=, <=] [ANY, ALL, SOME]`
Membership	Tests for membership in a list or subquery. `[NOT] IN`
Range	Tests for inclusion in a range. `[NOT] BETWEEN`
NULL	Tests for nulls. `IS NULL, IS NOT NULL`
EXISTS	Tests for existence of rows in a subquery. `[NOT] EXISTS`
LIKE	Specifies a test involving pattern matching. `[NOT] LIKE`
Compound	Specifies a combination of other conditions. `CONDITION [AND/OR] CONDITION`